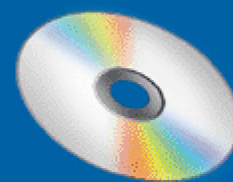
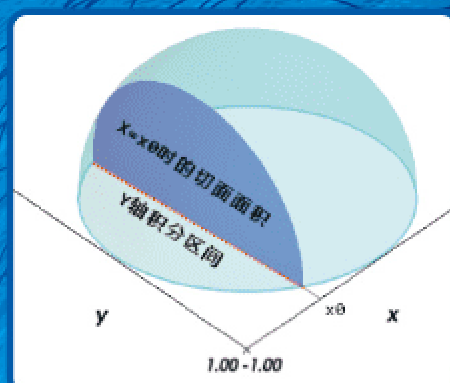
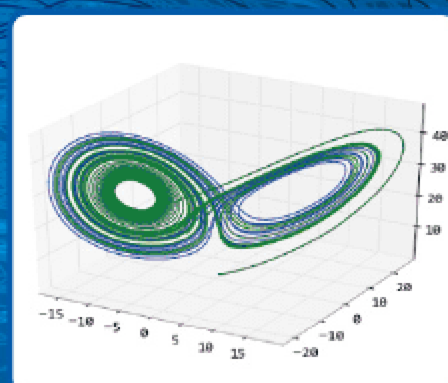
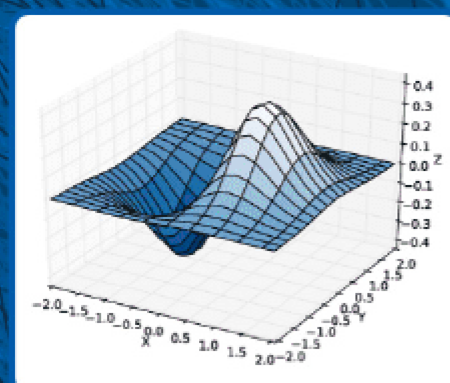
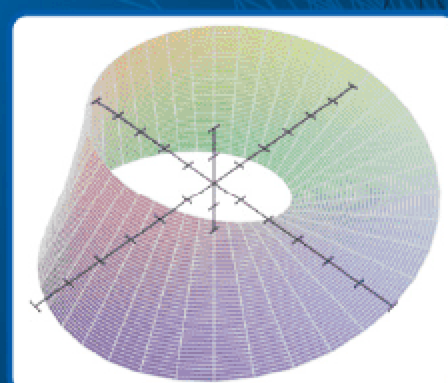
从 $e^{i\pi}+1$ 开始



Scientific Computing With Python

Python科学计算




张若愚 著

- ★ NumPy——快速处理数据
- ★ SymPy——符号运算好帮手
- ★ Traits——为Python添加类型定义
- ★ Chaco——交互式图表
- ★ Mayavi——更方便的可视化
- ★ OpenCV——图像处理和计算机视觉
- ★ 数字信号、滤波器、频域处理
- ★ 用C语言提高计算效率
- ★ SciPy——数值计算库
- ★ matplotlib——绘制精美的图表
- ★ TraitsUI——轻松制作用户界面
- ★ TVTK——数据的三维可视化
- ★ VPython——制作3D演示动画
- ★ 声音与视频数据处理
- ★ 动画模拟、分形几何

清华大学出版社

Python 科学计算

张若愚　著

清华大学出版社

北　京

内 容 简 介

本书介绍如何用 Python 开发科学计算的应用程序，除了介绍数值计算之外，还着重介绍如何制作交互式的 2D、3D 图像，如何设计精巧的程序界面，如何与 C 语言编写的高速计算程序结合，如何编写声音、图像处理算法等内容。书中涉及的 Python 扩展库包括 NumPy、SciPy、SymPy、matplotlib、Traits、TraitsUI、Chaco、TVTK、Mayavi、VPython、OpenCV 等，涉及的应用领域包括数值运算、符号运算、二维图表、三维数据可视化、三维动画演示、图像处理以及界面设计等。

书中以大量实例引导读者逐步深入学习，每个实例程序都有详尽的解释，并都能在本书推荐的运行环境中正常运行。此外，本书附有大量的图表和插图，力求减少长篇的理论介绍和公式推导，以便读者通过实例和数据学习并掌握理论知识。

图书在版编目(CIP)数据

Python 科学计算 / 张若愚　著. —北京：清华大学出版社，2012.1
ISBN 978-7-302-27360-8

Ⅰ. P…　Ⅱ. 张…　Ⅲ. 软件工具—程序设计　Ⅳ. TP311.56

中国版本图书馆 CIP 数据核字(2011)第 232994 号

责任编辑： 王　军　李维杰
装帧设计： 牛艳敏
责任校对： 成凤进
责任印制：
出版发行： 清华大学出版社　　**地　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：** 100084
社　总　机： 010-62770175　　**邮　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销： 全国新华书店
开　本： 185×260　　**印　张：** 39.75　　**字　数：** 941 千字
附光盘 1 张
版　次： 2012 年 1 月第 1 版　　**印　次：** 2012 年 1 月第 1 次印刷
印　数： 1~3000
定　价： 98.00 元

产品编号：

Preface

Python is rightfully viewed as a general purpose language, well suited for web development, system administration, and general purpose business applications. It's has earned this reputation well by powering web sites such as YouTube, installation tools integral to Red Hat's operating system, and large corporate IT systems from cloud cluster management to investment banking. Python has also established itself firmly in the world of scientific computing covering a wide range of applications from seismic processing for oil exploration to quantum physics. This breadth of applicability is significant because these seemingly disparate uses often overlap in important ways. Applications that can easily connect to databases publish information to the web, and efficiently carry out complex calculations are now critical in many industries. Python's primary strength is that it allows developers to build such tools quickly.

Python's scientific computing roots actually go quite deep. Guido van Rossum created the language while at CWI, the Center for Mathematics and Computer Science, in the Netherlands. As interest developed outside the center, others began to contribute. The first several Python workshops, starting in 1994, were held an ocean away at scientific institutions such as NIST (National Institute of Instruments and Technology), the US Geological Society, and LLNL (Lawrence Livermore National Laboratories), all science centric institutions. At the time, Python 1.0 had recently been released and the attendees were just beginning to hammer out the design of its mathematical tools. A decade and a half later, it is gratifying to see how far we have come both in the amazing capabilities of the tool set and the diversity of the community. It is somehow fitting that the first comprehensive book (that I know of) covering the primary scientific computing tools for Python is composed and published, another ocean away, in Chinese. Looking forward a decade and a half, I can hardly wait to see what we will all build together.

Guido, himself, was not a scientist or engineer. He sat squarely in the computer science branch of CWI and created Python to ease the pain of building system administration tools for the Amoeba operating system. At the time, the tools were being written in C. Python was to be the tool that "bridged the gap between shell scripting and C." Operating system tools are not even in the same neighborhood as matrix inversions or fast Fourier transforms, but, as the language emerged, scientists around the world were some of its earliest adopters. Guido had succeeded in creating an elegantly expressive language that coupled nicely with their existing C and Fortran code. And, in Guido, they had a language designer willing to listen and add critical features, such as complex numbers, specifically for the scientific community. With the creation of Numeric, the precursor to NumPy, Python gained a fast and powerful number crunching tool that solidified Python's role as a leading computational language in the coming decades.

For some, the term “scientific programming” conjures up visions of intricate algorithms described from “Numerical Recipes in C” or forged in late night programming sessions by graduate students. But the reality is the domain encompasses a much wider range of programming tasks from low level algorithms to GUI development with advanced graphics. This latter topic is too often underestimated in terms of importance and effort. Fortunately, Ruoyu Zhang has done us the service of covering all facets of the scientific programming in this book. Beginning with the foundational Numpy library the algorithmic toolboxes in SciPy he provides the fundamental tools for any scientific application. He then aptly covers the 2D plotting and 3D visualization libraries provided by matplotlib, chaco, and mayavi. Application and GUI development with Traits and Traits UI, and coupling to legacy C libraries through Cython, Weave, ctypes, and SWIG are well covered as well. These core tools are rounded out by coverage of symbolic mathematics with SymPy and various other useful topics.

It’s truly gratifying to see all of these topics aggregated into a single volume. It provides a one-stop shop that can lead you from the beginning steps to a polished and full featured application for analysis and simulation.

Eric Jones

2011/12/8

序

Python 理所当然地被视为一门通用的程序设计语言，非常适合于网站开发、系统管理以及通用的业务应用程序。它为诸如 YouTube 这样的网站系统、Red Hat 操作系统中不可或缺的安装工具以及从云管理到投资银行等大型企业的 IT 系统提供技术支持，从而赢得了如此高的声誉。Python 还在科学计算领域建立了牢固的基础，覆盖了从石油勘探的地震数据处理到量子物理等范围广泛的应用场景。Python 这种广泛的适用性在于，这些看似不同的应用领域通常在某些重要的方面是重叠的。易于与数据库连接、在网络上发布信息并高效地进行复杂计算的应用程序对于许多行业是至关重要的，而 Python 最主要的长处就在于它能让开发者迅速地创建这样的工具。

实际上，Python 与科学计算的关系源远流长。吉多 • 范罗苏姆创建这门语言，还是在他在荷兰阿姆斯特丹的国家数学和计算机科学研究学会(CWI)的时候。当时只是作为“课余”的开发，但是很快其他人也开始为之做出贡献。从 1994 年开始的头几次 Python 研讨会，都是在大洋彼岸的科研机构举行的。例如国家标准技术研究所(NIST)、美国地质学会以及劳伦斯利福摩尔国家实验室(LLNL)，所有这些都是以科研为中心的机构。当时 Python 1.0 刚刚发布，与会者们就已经开始打造 Python 的数学计算工具。10 多年过去了，我们欣喜地看到，我们在开发具有惊人能力的工具集以及建设多彩的社区方面做出了如此多的成绩。很合时宜的是，就我所知的第一本涵盖了 Python 的主要科学计算工具的综合性著作，在另一个海洋之遥的中国编著并出版了。展望今后的十几年，我迫不及待地想看到我们能共同创建出怎样的未来。

吉多他本人并不是科学家或工程师。他在 CWI 的计算机科学部门时，为了缓解为阿米巴(Amoeba)操作系统创建系统管理工具的痛苦，他创建了 Python。当时那些系统管理工具都是用 C 语言编写的。于是 Python 就成了填补 shell 脚本和 C 语言之间空白的工具。操作系统工具与计算逆矩阵或快速傅立叶变换是完全不同的领域，但是从 Python 诞生开始，世界各地的许多科学家就成了它最早期的采用者。吉多成功地创建了一门能与他们的 C 和 Fortran 代码完美结合的、具有优雅表现力的程序语言。并且，吉多是一位愿意听取建议并添加关键功能的语言设计师，例如支持复数就是专门针对科学领域的。随着 NumPy 的前身——Numeric 的诞生，Python 获得了一个高效且强大的数值运算工具，它巩固了在未来几十年中，Python 作为领先的科学计算语言的地位。

对于一些人来说，“科学计算编程”会让人联想起 *Numerical Recipes in C* 中描述的那些复杂算法，或是研究生们在深夜中努力打造程序的场景。但是真实情况所涵盖的范围更广泛——从底层的算法设计到具有高级绘图功能的用户界面开发。而后者的重要性却常常被忽视了。幸运的是在本书中，作者为我们介绍了科学计算编程所需的各个方面。从 NumPy 库和 SciPy 算法工具库的基础开始，介绍了任何科学计算应用程序所需的基本工具。然后，本书很恰当地介绍了二维绘图以及三维可视化库——matplotlib、Chaco、Mayavi。用 Traits 和 TraitsUI 进行应用程序和界面开发，以及用 Cython、Weave、ctypes 和 SWIG 等与传统的 C 语言库相互结合等

内容在书中也有很好的介绍。除了这些核心的工具之外，本书还介绍了使用 SymPy 进行数学符号运算以及其他的各种有用的主题。

所有这些主题都被汇编到一本书中真是一件令人欣喜的事情。本书所提供的一站式服务，能够指导读者从最初的入门直到创建一个漂亮的、全功能的分析与模拟应用程序。

Eric Jones
2011 年 12 月 8 日

关于序言作者

Eric Jones 是 Enthought 公司的 CEO，他在工程和软件开发领域拥有广泛的背景，指导 Enthought 公司的产品工程和软件设计。在共同创建 Enthought 公司之前，他在杜克大学电机工程学系从事数值电磁学以及遗传优化算法方面的研究，并获得了该系的硕士和博士学位。他教授过许多用 Python 做科学计算的课程，并且是 Python 软件基金会的成员。

关于 Enthought 公司

Enthought 是一家位于美国得克萨斯州首府奥斯汀的软件公司，主要使用 Python 从事科学计算工具的开发。本书中介绍的 NumPy、SciPy、Traits、TraitsUI、Chaco、TVTK 以及 Mayavi 均为该公司开发或维护的开源程序库。

前　言

Python 是一种面向对象的、动态的程序设计语言，具有非常简洁而清晰的语法，既可以用于快速开发程序脚本，也可以用于开发大规模的软件，特别适合于完成各种高层任务。

随着 NumPy、SciPy、matplotlib、ETS[①]等众多程序库的开发，Python 越来越适合于做科学计算。与科学计算领域最流行的商业软件 MATLAB 相比，Python 是一门真正的通用程序设计语言，比 MATLAB 所采用的脚本语言的应用范围更广泛，有更多程序库的支持，适用于 Windows 和 Linux 等多种平台，完全免费并且开放源码。虽然 MATLAB 中的某些高级功能目前还无法替代，但是对于基础性、前瞻性的科研工作和应用系统的开发，完全可以用 Python 来完成。

本书介绍如何用 Python 开发科学计算的应用程序，除了介绍数值计算之外，还着重介绍了如何制作交互式二维、三维图像，如何设计精巧的程序界面，如何与 C 语言编写的高速计算程序结合，如何编写声音、图像处理算法等内容。

由于 Python 的相关资源非常多，本书不可能全部涉及，相信读者在掌握本书所介绍的一些相关知识之后，只要充分利用互联网的搜索功能，就一定能够很快地找到合适的 Python 解决方案。此外，由于绝大多数 Python 资源都开放源代码，因此读者将会很容易地对感兴趣的内容进行深度挖掘和研究。

本书适合于工科高年级本科生、研究生、工程技术人员以及计算机开发人员阅读。实例篇以信号处理为主，通过简单易懂的 Python 源程序，实际演示信号处理的一些基础知识和原理，因此特别适合于相关专业的学生作为扩展视野的补充阅读教材。

阅读本书的读者需要掌握 Python 语言的一些基础知识，下面是一个“自我检测列表”，如果读者熟悉下述内容，阅读本书的实例源代码就应该没有困难。此外由于 Python 程序简单易读，即使读者没有接触过 Python，也可以边阅读本书边通过其他书籍或免费教程学习 Python。

- 基本语法：库的载入(import)、循环(for、while)、判断(if)、函数定义(def)
- 基本数据类型的用法：列表(list)、字典(dict)、元组(tuple)、字符串
- 面向对象的基本语法：类(class)、继承
- C 语言编程的基础知识[②]

有关 Python 语言的基础知识，可以参考啄木鸟社区的 Python 图书简介。



http://wiki.woodpecker.org.cn/moin/PyBooks
啄木鸟社区的 Python 图书概览

本书所有演示程序，均在 Windows XP 系统下采用 Python(x,y)通过测试。如果读者觉得安装众多的 Python 程序库很麻烦，不妨下载安装 Python(x,y)，或者直接使用本书所附光盘中的 Python(x,y)安装程序。

---

① 全称为 Enthought Tool Suite，是 Enthought 公司开发的开源科学计算应用程序开发包。

② 为了提高程序的运行效率，有时需要使用 C 语言编写 Python 的扩展模块，第 16 章“用 C 语言提高计算效率”中介绍的内容需要读者熟悉 C 语言编程。

目　　录

第 1 章

软件包的安装和介绍

1.1 Python 简介

Python 是一种解释型、面向对象、动态的高级程序设计语言。自从 20 世纪 90 年代初 Python 语言诞生至今，它逐渐被广泛应用于处理系统管理任务和 Web 编程。目前 Python 已经成为最受欢迎的程序设计语言之一。2011 年 1 月，它被 TIOBE 编程语言排行榜评为 2010 年度语言。

由于 Python 语言的简洁、易读以及可扩展性，在国外用 Python 做科学计算的研究机构日益增多，一些知名大学已经采用 Python 教授程序设计课程。例如麻省理工学院的计算机科学及编程导论课程[①]就使用 Python 语言讲授。众多开源的科学计算软件包都提供了 Python 的调用接口，例如著名的计算机视觉库 OpenCV、三维可视化库 VTK、医学图像处理库 ITK。而 Python 专用的科学计算扩展库就更多了，例如如下 3 个十分经典的科学计算扩展库：NumPy、SciPy 和 matplotlib，它们分别为 Python 提供了快速数组处理、数值运算以及绘图功能。因此 Python 语言及其众多的扩展库所构成的开发环境十分适合工程技术、科研人员处理实验数据、制作图表，甚至开发科学计算应用程序。

说起科学计算，首先会被提到的可能是 MATLAB。然而除了 MATLAB 的一些专业性很强的工具箱目前还无法替代之外，MATLAB 的大部分常用功能都可以在 Python 世界中找到相应的扩展库。和 MATLAB 相比，用 Python 做科学计算有如下优点：

- 首先，MATLAB 是一款商用软件，并且价格不菲。而 Python 完全免费，众多开源的科学计算库都提供了 Python 的调用接口。用户可以在任何计算机上免费安装 Python 及其绝大多数扩展库。
- 其次，与 MATLAB 相比，Python 是一门更易学、更严谨的程序设计语言。它能让用户编写出更易读、易维护的代码。
- 最后，MATLAB 主要专注于工程和科学计算。然而即使在计算领域，也经常会遇到文件管理、界面设计、网络通信等各种需求。而 Python 有着丰富的扩展库，可以轻易完成各种高级任务，开发者可以用 Python 实现完整应用程序所需的各种功能。

例如，笔者在一个模拟控制系统的项目中，完全用 Python 实现了系统模拟及算法优化，

---

① 此课程的英文全称为 Introduction to Computer Science and Programming，它是麻省理工学院的开放课程之一，读者可以在 MIT 开放课程网(http://ocw.mit.edu)上找到此课程的全部视频及课件。

并在此基础上实现了应用程序必需的文档和数据库管理、用户界面设计、与机器设备及其他软件进行通信等功能。最后，整个应用程序可以随意安装到不同的计算机上，而不受任何商用软件的使用条款限制。

1.2　安装软件包

和 MATLAB 等商用软件不同，Python 的众多扩展库由许多社区分别维护和发布，因此要一一将它们收集齐全并安装到计算机中是一件十分耗费时间和精力的事情。本书介绍两个科学计算用的 Python 集成软件包。读者只需要下载并执行一个安装程序，就能安装好本书涉及的所有扩展库。

1.2.1　Python(x,y)

Python(x,y)收集了众多的扩展库、文档和教程。在本书所附的光盘中提供了 Python(x,y) 2.6.6 的安装程序，为了保证能正确运行本书的所有实例程序，推荐读者以完全安装模式进行安装。Python(x,y)的版本号与它所使用的 Python 版本号相同。

http://www.pythonxy.com
Python(x,y)官方网址

为了确保本书的所有实例程序都能正常运行，请读者参照图 1-1 修改安装选项。将安装模式修改为完全安装，并将 Python(x,y)的安装路径改为“c:\pythonxy”。否则 Python 将可能无法正确调用 MinGW 编译扩展模块。请读者在安装结束之后，确认下列路径，在今后的章节中会经常用到它们：

c:\python26 Python 2.6 的安装路径，所有扩展库都可以在它的子目录“lib\site-packages”下找到
c:\pythonxy\doc 众多扩展库的说明文档和演示程序
c:\pythonxy\mingw MinGW C/C++编译器，在介绍用 C 语言编写扩展模块时会用到它
c:\pythonxy\swig 自动生成扩展模块接口的工具，在介绍用 C 语言编写扩展模块时会用到它

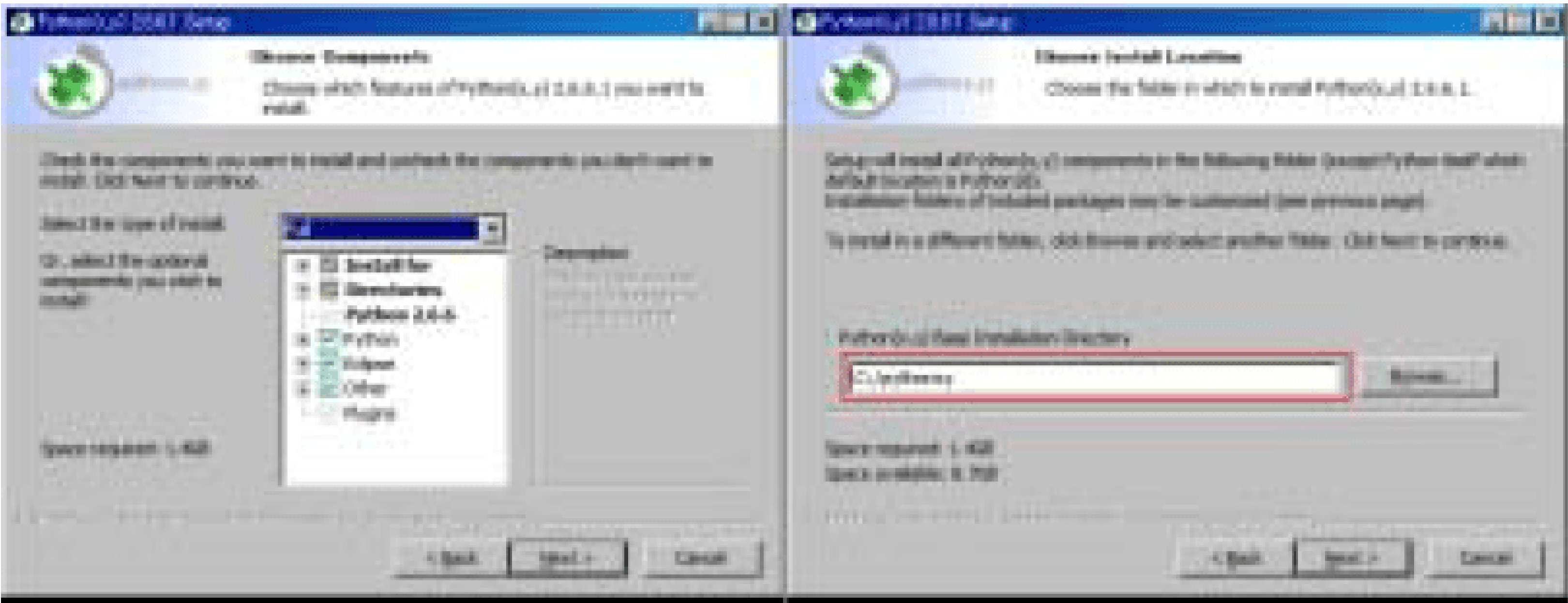


图 1-1　选择“Full”进行完全安装，并将 python(x, y)”的安装路径设置为“C:\pythonxy”

如果使用默认的安装路径，那么“pythonxy”目录会被安装到“c:\Program Files”下。

Python(x,y)提供了如图 1-2 所示的启动程序，从中可以快速启动各种工具，以及打开文档教程所在的文件夹。

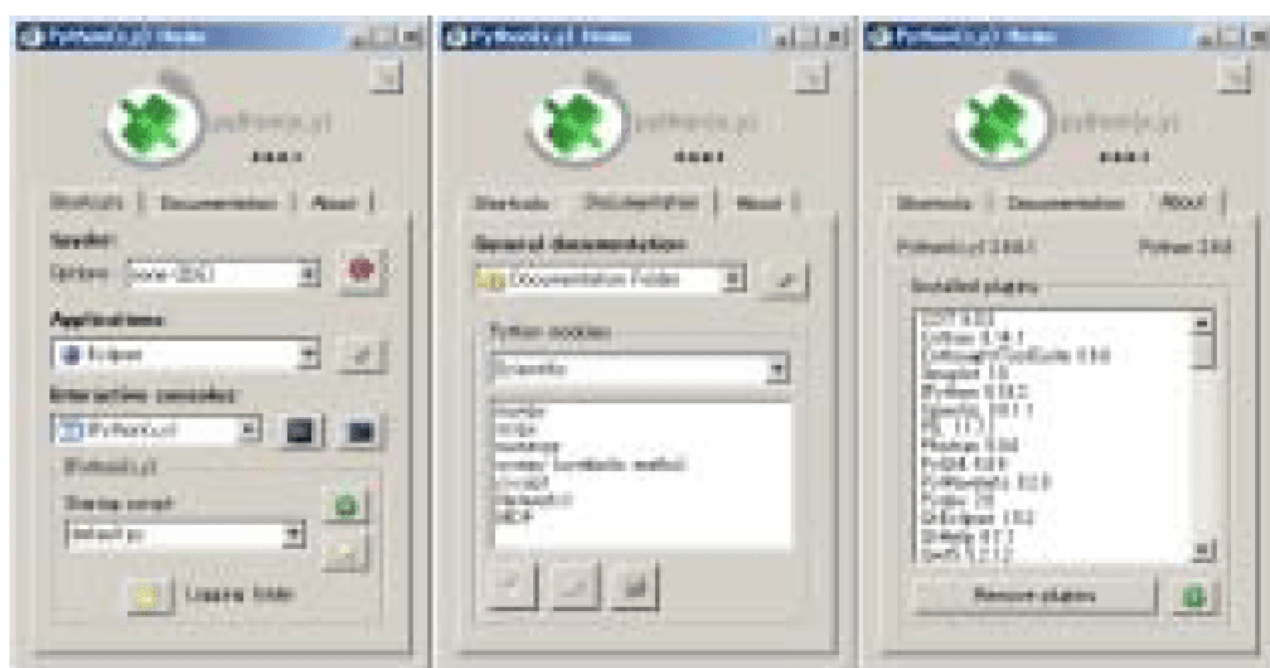


图 1-2 Python(x,y)的启动画面

界面中的 3 个选项卡如下：

- Shortcuts：启动各种应用程序，包括 Eclipse、Mayavi、Spyder 集成开发环境、IPython 交互式命令行等。
- Documentation：打开扩展库的文档，这里列出的文档不是很全面，推荐读者直接打开“c:\pythonxy\doc”文件夹查找文档。
- About：查看所安装的扩展库的版本信息。

1.2.2 Enthought Python Distribution(EPD)

EPD 是一个商用的 Python 发行版本，同样包括了众多的科学软件包，而且作为教学使用是免费的，只需要提供一个教育单位的邮件地址，就可以收到 EPD 的教育版的下载地址。



http://www.enthought.com/products/getepd.php
EPD 的官方下载地址

1.3 方便的开发工具

本节介绍几个在开发和调试程序时经常会用到的工具软件，熟练掌握它们的用法能够起到事半功倍的效果。为了展示工具软件的功能，本节以一些扩展库作为演示。读者可以暂时忽略这些扩展库的用法，在后续的章节中会对它们进行详细介绍。

1.3.1 IPython

IPython 是 Python 的一个交互式命令行工具，与 Python 自带的命令行相比，它更容易使用，功能也更强大。它支持语法高亮、自动补全、自动缩进，并且内置了许多有用的功能和函数。

如果读者安装了 Python(x,y)，就可以从它的启动界面中运行 IPython，如图 1-3 所示。



图 1-3 通过 Python(x,y) Home 启动 IPython 的各种选项

从下拉列表中选择想运行的命令行配置，然后单击右侧的❶或❷按钮运行所选的命令行配置。其中，“Python”选项运行 Python 自带的命令行工具，而“IPython(x,y)”、“IPython(Qt)”、“IPython(wxPython)”、“IPython(mlab)”和“IPython(sh)”等几个选项，分别使用表 1-1 所示的参数来运行 IPython。

表 1-1 运行 IPython 的其他选项及对应参数

选　项	参　数
IPython(x,y)	-pylab -p xy
IPython(Sh)	-q4thread -p sh
IPython(Qt)	-q4thread
IPython(wxPython)	-wthread
IPython(mlab)	-wthread -p mlab
IPython(sh)	-q4thread -p sh

单击❶按钮将用一个名为 Console 的软件启动命令行，此软件使用 Windows 的窗口界面封装命令行界面，并且具有标签页功能。单击❷按钮将用 Windows 自带的命令行界面进行启动。

如果运行“IPython(x,y)”，在启动 IPython 之后将自动运行一个名为“default.py”脚本文件。此脚本默认执行以下函数库的导入操作：

```
import numpy
import scipy
from numpy import *
```

为了与 NumPy、SciPy 社区的推荐导入方式一致，请单击按钮❸，在打开的文件夹中添加一个名为“myimports.py”的文件，并在其中添加如下几行程序代码：

```
import numpy as np
import scipy as sp
import pylab as pl
```

此后运行“IPython(x,y)”的时候请选择“myimports.py”作为启动脚本。

如果要在命令行中和 matplotlib、TraitsUI、Mayavi 等图形界面程序进行交互，就需要给 IPython 传递“-wthread”、“-q4thread”或“-pylab”参数。当使用“-pylab”参数时，在调用 matplotlib 库的绘图函数进行绘图时，将立即显示图表。

下面我们以 matplotlib 绘图为例，实际操作一下。请读者选择“myimports.py”作为启动脚本，并运行 IPython(x,y)命令行。然后在命令行中输入下面的语句，如果一切顺利，将会立即显示出如图 1-4 所示的正弦波形。

```
>>> x = np.linspace(0, 4*np.pi, 100)
>>> pl.plot(x, np.sin(x))
```

这里的 np 和 pl 是运行“myimports.py”之后的结果，它们分别表示 NumPy 和 pylab 模块。由于 IPython(x,y)的命令行参数中有“-pylab”，pylab 模块中的所有符号也会被载入到当前的名称空间中，因此也可以用下面的程序绘制正弦波形：

```
>>> x = linspace(0, 4*pi, 100)
>>> plot(x, sin(x))
```

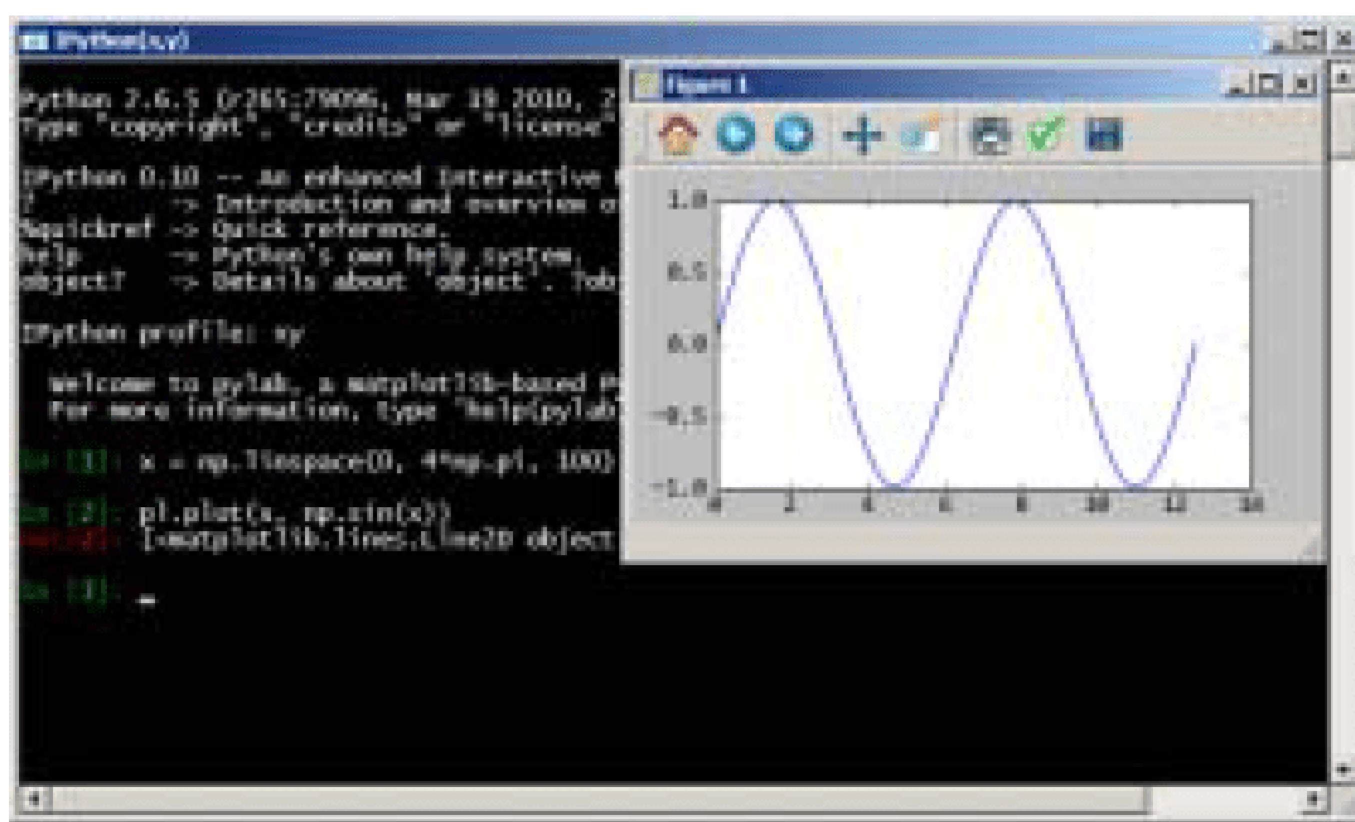


图 1-4　使用 IPython 交互式地绘制正弦波



本书中所有在 IPython 下输入的代码都用“>>>”作为提示符，而不使用 IPython 命令行的默认提示符“In[..]”。

在 IPython 中，可以很方便地使用如下功能：

- 自动补全：输入一部分文字之后按 Tab 键，IPython 将列出所有补全信息。用此功能可以快速输入对象的属性名或者进行文件名补全。
- 查看文档：输入需要查看文档的函数名，然后在后面添加一个或两个问号。“?”表示查看函数的文档，“??”表示查看其 Python 源代码。如果函数不是用 Python 编写的，就看不到其源代码。

- 执行 Python 程序：用 run 命令运行指定的 Python 程序文件。默认是在一个新的环境中运行程序，当程序退出时将程序运行环境中的对象复制到 IPython 环境中。如果运行 run 命令时添加“-i”参数，在 IPython 的当前环境中执行程序，程序即可直接访问 IPython 环境中的对象。
- 执行剪切板中的程序：运行 paste 命令将在 IPython 环境中运行剪贴板中的程序代码，它会自动删除代码中的提示符“>>>”。运行“paste foo”将把剪切板中的内容复制到变量 foo 中。变量 foo 是 IPython 提供的 SList 列表类型，它提供了很多对其内容进行操作的方法。
- 执行系统命令：在要执行的系统命令之前添加一个“!”符号。例如，如果执行“!test.py”，那么操作系统会运行“test.py”文件。和 run 命令不同，“test.py”将在另外的进程中运行。

在 IPython 中使用 run 命令运行 Python 程序能够提高调试程序的速度。因为每次运行用户程序时，IPython 环境没有初始化，已经载入的模块不需要重新载入，对于 NumPy、SciPy 和 matplotlib 这样比较庞大的库，能节省不少载入时间。下面是一个例子：



speedup test.py
载入 NumPy、SciPy 和 pylab 库并绘制一个简单的频率扫描波

```
import numpy as np
from scipy import signal
import pylab as pl

t = np.linspace(0, 10, 1000)
x = signal.chirp(t, 5, 10, 30)
pl.plot(t, x)
pl.show()
```

此程序计算频率扫描波并使用图表来显示，如果通过双击或者在命令行中输入文件名来直接运行此程序，需要等几秒钟才能看到结果。如果先在“speedup_test.py”所在的文件夹启动 IPython，然后运行：

```
>>> run speedup_test.py
```

第一次运行时由于同样需要载入所需的模块，因此所需的时间和直接运行没有差别，但是再次运行时由于不需要载入模块，因此结果可以立即显示出来。此后还可以在 IPython 中输入程序，查看变量的值或者修改界面的各种属性，例如：

```
>>> x[:5] #查看扫描波的前 5 个值
array([ 1.        ,  0.95076436,  0.80716308,  0.58242516,  0.29822084])
>>> pl.gca().lines[0].set_color("r") #设置曲线颜色为红色
>>> pl.draw() #刷新界面
```

Python 的模块缓存功能虽然可以加速程序运行，但是也会带来一些问题。例如，如果有一个名为“mymodel.py”的模块，在“usemodel.py”中导入 mymodel 模块，那么只有在第一次运行“run usemodel.py”时会载入 mymodel 模块，以后即使“mymodel.py”文件发生改变，Python 也不会重新载入最新的模块。如果要修改调试“mymodel.py”中的程序，就需要在“usemodel.py”中添加如下两行代码：

```
import mymodel
reload(mymodel)
```

这样一来，每次运行“usemodel.py”时都会强制重新载入 mymodel 模块。



mymodel.py, usemodel.py
使用 reload 函数重新载入模块

IPython 还有很强大的调试功能，例如下面的程序使用 *sin(x cos(x))*计算一个长度为 10000 的数组，并且调用 imshow()将此数组显示成一个二维图像。



ipython_debug.py
用 IPython 调试程序中的错误

```
import pylab as pl
import numpy as np

def test_debug():
    x = np.linspace(1, 50, 10000)
    img = np.sin(x*np.cos(x))
    pl.imshow(img)
    pl.show()

test_debug()
```

但是由于程序中有错误，在 IPython 中运行它时，会出现很长一串错误信息。下面给出的是错误信息的最后一部分，我们看到抛出异常的是“image.pyc”中的 set_data()函数，它是扩展模块中的一个函数，错误信息的意思是——作为图像数据的数组的维数不正确。

```
>>> run ipython_debug.py
[[省略]]
    C:\Python26\lib\site-packages\matplotlib\image.pyc in set_data(self, A)
    298         if (self._A.ndim not in (2, 3) or
    299             (self._A.ndim == 3 and self._A.shape[-1] not in (3, 4))):
--> 300             raise TypeError("Invalid dimensions for image data")
    301
    302         self._imcache =None
    TypeError: Invalid dimensions for image data
    WARNING: Failure executing file: <ipython_debug.py>
```

为了找到程序中出错的位置，在 IPython 中输入 debug 命令，进入调试状态，并显示出调用堆栈的当前位置：

```
>>> debug
> c:\python26\lib\site-packages\matplotlib\image.py(300)set_data()
    299             (self._A.ndim == 3 and self._A.shape[-1] not in (3, 4))):
--> 300             raise TypeError("Invalid dimensions for image data")
    301
ipdb>
```

调试状态的提示符为“ipdb”，输入“h”命令可以查看调试状态下能用的所有命令，输入“h 命令名”可以查看命令的详细说明。连续执行多次“u”命令，沿着调用堆栈往上溯源，直到找到“ipython_debug.py”中出错的那一行：

```
ipdb> u
> c:\zhang\pydoc\source\examples\01-intro\ipython_debug.py(7)test_debug()
      6     img = np.sin(x*np.cos(x))
----> 7     pl.imshow(img)
      8     pl.show()
```

由错误信息可知数组 img 的维数不对。查看表示数组维数的 ndim 属性，发现 img 是一维数组，而 imshow()的参数应该是二维数组：

```
ipdb> img.ndim
1
```

输入“q”命令结束调试，并编辑“ipython_debug.py”，在调用 imshow()之前添加下面一行代码：

```
img.shape = 100, -1
```

然后再重新执行程序，这次就可以看到表示二维数组的图像了。

```
>>> run ipython_debug.py
```

1.3.2 Spyder

Spyder 是 Python(x,y)的作者为它开发的一个简单的集成开发环境。和其他的 Python 开发环境相比，它最大的优点就是模仿 MATLAB 的“工作空间”的功能，可以很方便地观察和修改数组的值。图 1-5 是 Spyder 的界面截图。



http://code.google.com/p/spyderlib

Spyder 项目的地址

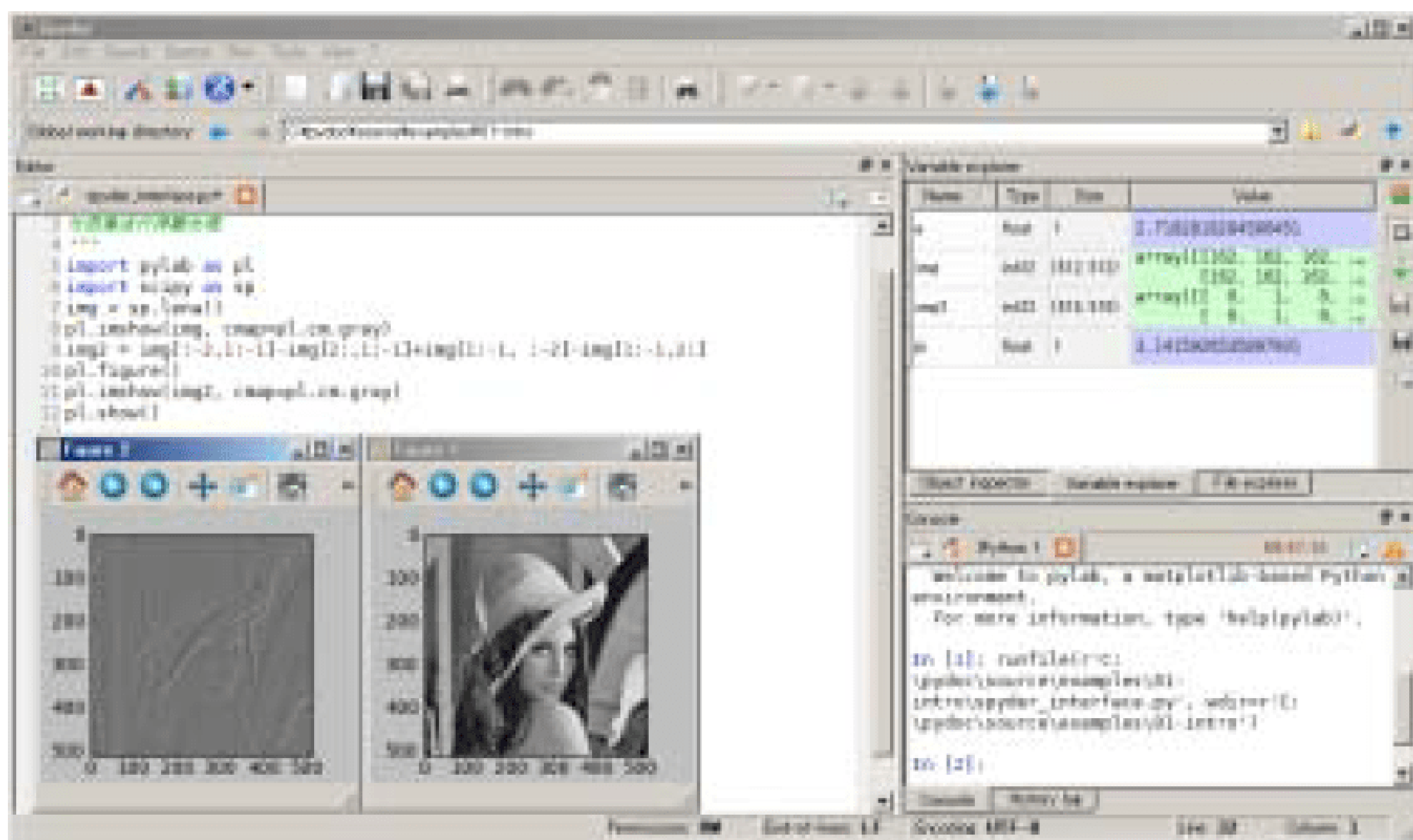


图 1-5　在 Spyder 中执行图像处理的程序

Spyder 的界面由许多窗格构成，用户可以根据自己的喜好调整它们的位置和大小。当多个窗格出现在一个区域时，将使用标签页的形式显示。例如在图 1-5 中，可以看到"Editor"、"Object inspector"、"Variable explorer"、"File explorer"、"Console"、"History log"以及两个显示图像的窗格。在 View 菜单中可以设置是否显示这些窗格。表 1-2 中列出了 Spyder 的主要窗格及其作用：

表 1-2　Spyder 的主要窗格及其作用

窗 格 名 称	作　用
Editor	编辑程序，可用标签页的形式编辑多个程序文件
Console	在别的进程中运行的 Python 控制台
Variable explorer	显示 Python 控制台中的变量列表
Object inspector	查看对象的说明文档和源程序
File explorer	文件浏览器，用于打开程序文件或者切换当前路径

按 F5 键将运行当前编辑器中的程序。第一次运行程序时，将弹出一个如图 1-6 所示的运行配置对话框。在此对话框中可以对程序的运行进行如下配置：

- Command line options：输入程序的运行参数。
- Working directory：输入程序的运行路径。
- Execute in current Python or IPython interpreter：在当前的 Python 控制台中运行程序。程序可以访问此控制台中的所有全局对象，控制台中已经载入的模块不需要重新载入，因此程序的启动速度较快。
- Execute in a new dedicated Python interpreter：新开一个 Python 控制台并在其中运行程序，程序的启动速度较慢，但是由于新控制台中没有多余的全局对象，因此更接近实际的运行情况。当选择此项时，还可以选中"Interact with the Python interpreter after

execution”复选框，这样当程序结束运行时，控制台进程将继续运行，因此可以通过它查看程序运行之后的所有全局对象。此外，还可以在“Command line options”中输入新控制台的启动参数。

运行配置对话框只会在第一次运行程序时出现，如果想修改程序的运行配置，可以按 F6 键打开运行配置对话框。

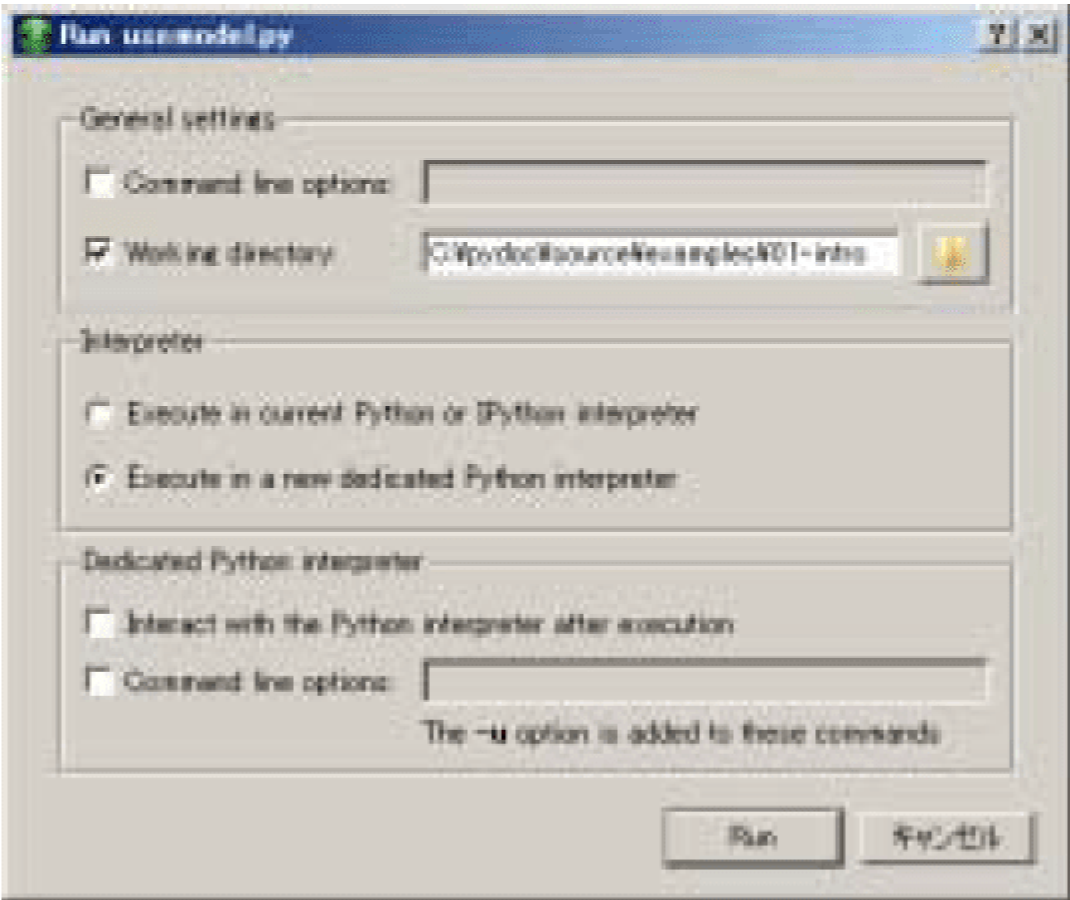


图 1-6　运行配置对话框

控制台中的全局对象可以在“Variable explorer”窗格中找到。此窗格支持数值、字符串、元组、列表、字典以及 NumPy 数组等对象的显示和编辑。图 1-7(左)是“Variable explorer”窗格的截图，其中列出了当前控制台中的变量名、类型、大小以及内容。右击变量名，弹出对此变量进行操作的菜单。在菜单中选择 Edit 选项，弹出图 1-7(右)所示的数组编辑窗口。此编辑窗口中，单元格的背景颜色直观地显示了数值的大小。

当有多个控制台运行时，“Variable explorer”窗格显示当前控制台中的全局对象。

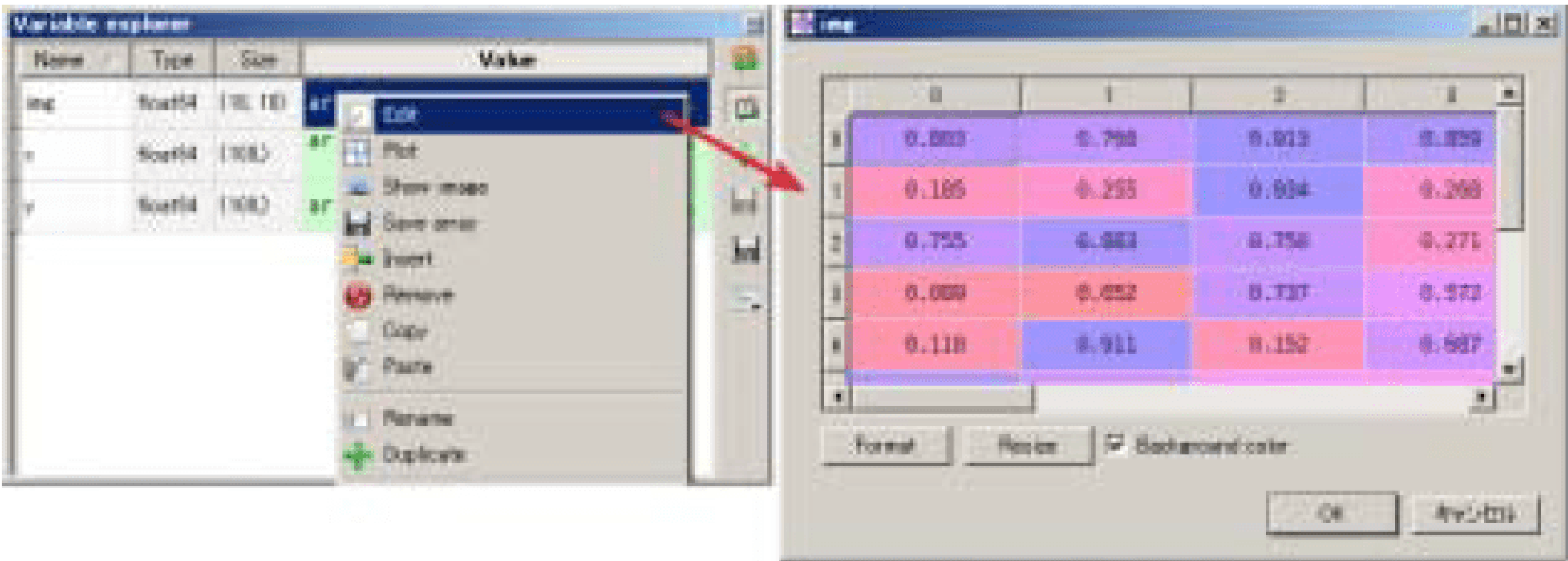


图 1-7　使用“Variable explorer”窗格查看和编辑数组的内容

选择菜单中的 Plot 选项，将弹出如图 1-8 所示的绘图窗口。在绘图窗口的右键菜单中选择“Parameters”，将弹出一个编辑绘图对象的对话框。图 1-8 中使用此对话框修改了曲线的颜色和线宽。

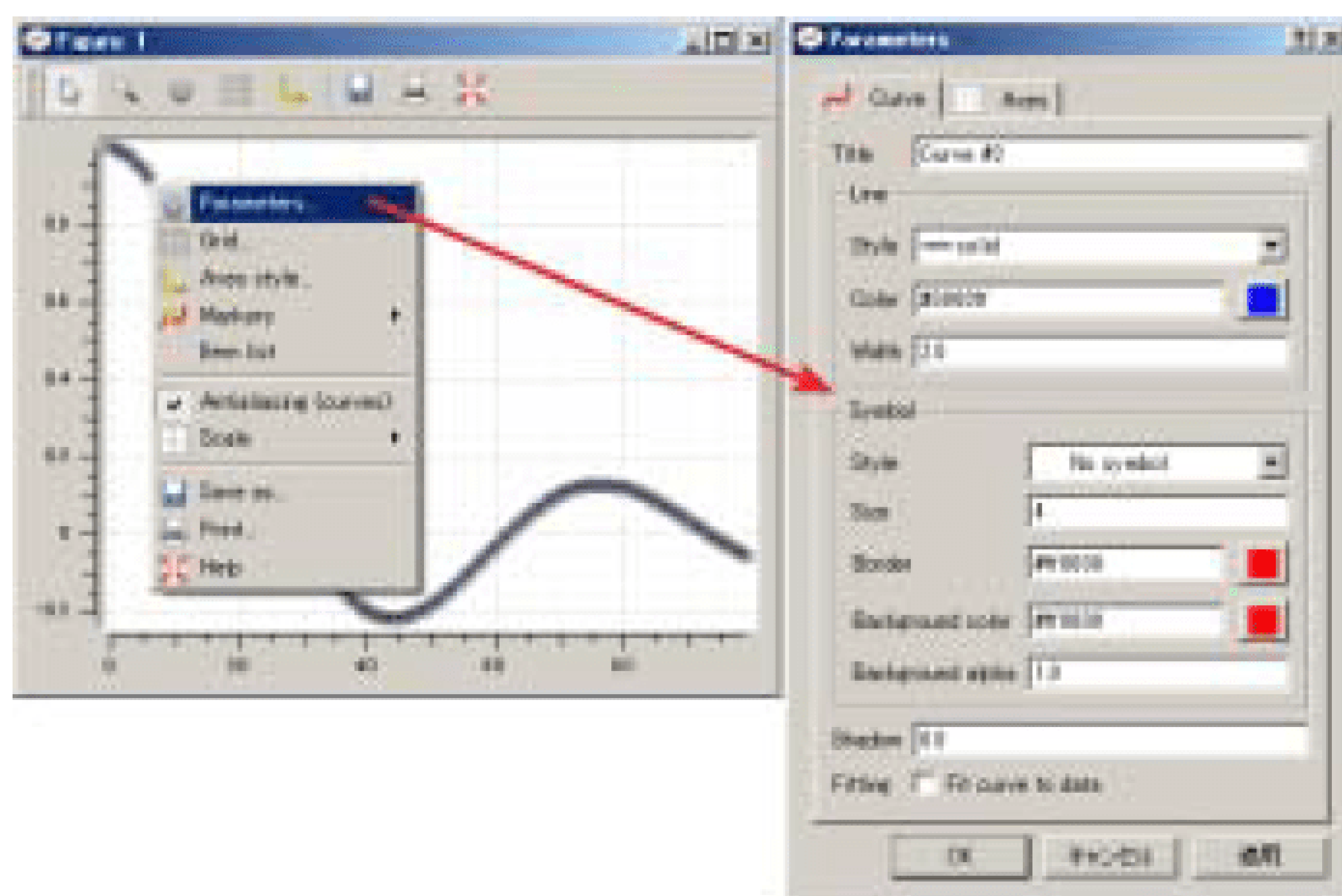


图 1-8　在“Variable explorer”窗格中将数组绘制成曲线

Spyder 的功能比较多，这里仅介绍一些常用的功能和技巧：

- 默认配置下，“Variable explorer”窗格中不显示以大写字母开头的变量，可以单击工具栏中的配置按钮(最后一个按钮)，在菜单中取消“Exclude capitalized references”的选中状态。
- 在控制台中，可以按 Tab 按键进行自动补全。在变量名之后输入“?”，可以在“Object inspector”窗格中查看对象的说明文档。此窗格的 Options 菜单中的“Show source”选项可以开启显示函数的源程序。
- 可以通过“Working directory”工具栏修改工作路径，用户程序运行时，将以此工作路径作为当前路径。例如我们只需要修改工作路径，就可以用同一个程序处理不同文件夹下的数据文件。
- 在程序编辑窗口中按住 Ctrl 键，并单击变量名、函数名、类名或模块名，可以快速跳转到定义位置。如果是在别的程序文件中定义的，将打开此文件。在学习一个新模块的用法时，我们经常需要查看模块中的某个函数或类是如何实现的，使用此功能可以帮助我们快速查看和分析各个模块的源程序。例如下面的程序从不同的扩展库载入了一些模块和类。用 Spyder 打开此文件，按住 Ctrl 键，并单击 signal、pl、HasTraits、Instance、View、Item、lfilter、plot、title 等，将打开定义它们的程序文件，并跳转到相应的行。



gotodefine.py
测试定义跳转功能

```
from scipy import signal
import pylab as pl
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item

signal.lfilter
pl.plot
pl.title
```

1.3.3 Wing IDE 101

Wing IDE 是一个功能强大的 Python 集成开发环境，它的专业版是商用软件，但是也提供了一个免费的简装版本 Wing IDE 101。



http://www.wingware.com/downloads/wingide-101
Wing IDE 101 的下载地址

和 Spyder 一样，在 Wing IDE 中只需要按住 Ctrl 键并同时单击函数名或类名，就能直接跳转到定义它的位置。此外，Wing IDE 还有不错的调试功能。在程序中设置断点之后，单击 Debug 按钮就可以进入调试运行模式。当运行到断点之后，程序将暂停运行。读者可以用 Wing IDE 打开下面的程序，并将光标移到“self.count += 1”一行，按 F9 键添加断点，然后按 F5 键开始调试程序。



wingide_debug.py
测试 Wing IDE 的断点调试功能

图 1-9 是调试程序时的界面截图。程序执行之后会显示出一个小窗口，其中有一个名为“Click Me”的按钮，单击它将调用程序中的_button_fired()，遇到断点从而暂停程序运行。此时可以观察程序的调用堆栈(Call stack)和堆栈数据(Stack Data)。

在主窗口左侧的“Stack Data”窗格中，显示了 locals 和 globals 两个字典，它们分别是当前执行环境下的全局变量和当前堆栈位置中的局部变量。下半部分显示了被选中的名为 self 的局部变量的内容。在主窗口下方的“Call Stack”窗格中显示了执行到断点处的调用堆栈，其中堆栈的顶部，即最下面一行被选中。可以用鼠标选中堆栈中的其他调用点，程序编辑窗格和“Stack Data”窗格中的内容也随之发生变化。通过这种方法可以观察堆栈中的所有局部变量，了解运行到断点处的整个调用过程，并查看与其相关的源程序。

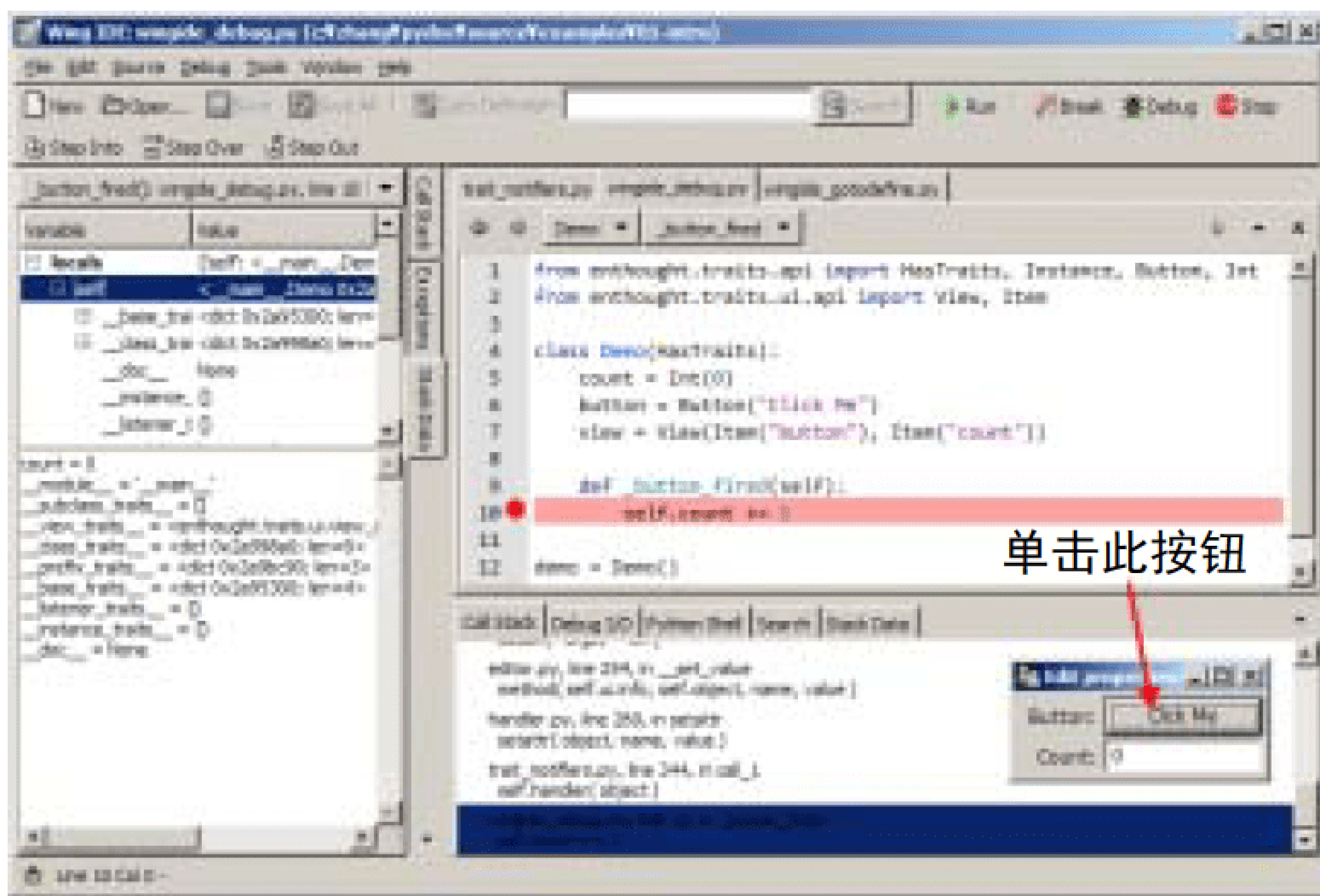


图 1-9　用 Wing IDE 101 调试程序

1.4　函数库介绍

Python 的科学计算功能由众多的扩展库协作完成。在本书的后续章节中，将对下列扩展库进行详细介绍。

1.4.1　数值计算库

NumPy 为 Python 带来了真正的多维数组功能，并且提供了十分丰富的对数组进行处理和运算的函数集。它对常用的数学函数进行数组化，使这些数学函数能直接对数组进行运算，将本来需要在 Python 中进行的循环运算，转移到高效率的库函数中，充分利用这些函数能明显地提高程序的运算速度。

SciPy 则在 NumPy 的基础之上添加了许多科学计算的函数库，其中一些函数是通过对久经考验的 Fortran 数值计算库进行封装实现的，例如：

- 线性代数使用 LAPACK 库
- 快速傅立叶变换使用 FFTPACK 库
- 常微分方程求解使用 ODEPACK 库
- 非线性方程组求解以及最小值求解等使用 MINPACK 库



http://www.scipy.org
SciPy 官方网址

有了这两个库，Python 就有几乎和 MATLAB 一样的数据处理能力了。此外，SciPy 中的 Weave 模块能在 Python 程序中直接嵌入 C++程序，进一步提高程序的运算速度。

1.4.2 符号计算库

SymPy 是一套数学符号运算的扩展库，虽然它目前还没有到达 1.0 版本，但是已经足够好用，可以帮助我们进行公式推导，做一些简单的符号运算工作。



http://code.google.com/p/sympy
SymPy 官方网址

1.4.3 界面设计

Python 可以使用多种界面库编写 GUI 程序，例如以 TK 为基础的 Tkinter、以 wxWidgets 为基础的 wxPython、以 QT 为基础的 pyQt4 等界面库。但是使用这些界面库编写 GUI 程序仍然是一件十分繁杂的工作。为了让读者不在界面设计上耗费大量精力，从而能把注意力集中到数据处理上，本书详细介绍了如何使用 Traits 库设计图形界面程序。



http://code.enthought.com/projects/traits
Traits 官方网址

Traits 库分为 Traits 和 TraitsUI 两大部分，使用 Traits 能对 Python 对象的属性进行类型定义，并为其添加初始化、校验、代理、事件处理等诸多功能。

TraitsUI 库基于 Traits 库，使用 MVC(模型—视图—控制器)模式快速地定义用户界面，在最简单的情况下，甚至不需要写一句界面相关的代码，就可以通过 Traits 的属性定义获得一个可用的图形界面。使用 TraitsUI 库编写的程序自动支持 wxPython 和 pyQt 界面库。

1.4.4 绘图与可视化

matplotlib 和 Chaco 是两个很优秀的二维绘图库。matplotlib 库能够快速地绘制精美的图表、以多种格式输出，并且带有简单的三维绘图功能。而 Chaco 则以 Traits 为基础，能够很方便地编写出交互式图表控件，并嵌入到用 TraitsUI 编写的界面程序中。



http://code.enthought.com/projects/chaco
Chaco 官方网址



http://matplotlib.sourceforge.net
matplotlib 官方网址

TVTK 库对标准的 VTK 库用 Traits 进行了封装，如果要在 Python 中使用 VTK，用 TVTK 是最方便的选择。Mayavi 则在 TVTK 的基础上添加了一套面向应用的方便工具，它既可以单独作为三维可视化程序使用，也可以很方便地嵌入到用 TraitsUI 编写的界面程序中。



http://code.enthought.com/projects/mayavi
Mayavi 官方网址

VTK(Visualization Toolkit)

视觉化工具库(Visualization Toolkit，VTK)是一个开放源码、跨平台、支援平行处理(VTK 曾用于处理大小近乎 1 个 PB 的资料，其平台为美国 Los Alamos 国家实验室的具 1024 个处理器的大型系统)的图形应用函数库。2005 年曾被美国陆军研究实验室用于即时模拟俄罗斯制反导弹战车 ZSU23-4 受到平面波攻击的情形，其计算节点高达 25 万个之多。

此外，使用 VPython 库能够快速、方便地制作三维动画演示，使数据更有说服力。



http://vpython.org
VPython 官方网址

1.4.5 图像处理和计算机视觉

OpenCV 最初是由英特尔公司开发的一套开源的跨平台计算机视觉库，可用于开发实时的图像处理、计算机视觉以及模式识别程序。它有多套 Python 的调用接口，本书将以其中的 pyOpenCV 为例介绍 OpenCV 的一些基础知识。pyOpenCV 库不但很全面地对 OpenCV 的各种函数和类进行了封装，而且能在 OpenCV 的图像对象和 NumPy 数组之间进行互换。这样便同时扩展了 NumPy 的图像处理能力以及 OpenCV 的数组处理能力。



http://code.google.com/p/pyopencv/
pyopencv 项目的地址

第 2 章

NumPy——快速处理数据

标准的 Python 中用列表(list)保存一组值，可以当作数组使用。但由于列表的元素可以是任何对象，因此列表中保存的是对象的指针。这样一来，为了保存一个简单的列表[1,2,3]，就需要有三个指针和三个整数对象。对于数值运算来说，这种结构显然比较浪费内存和 CPU 计算时间。

此外 Python 还提供了 array 模块，它所提供的 array 对象和列表不同，能直接保存数值，和 C 语言的一维数组类似。但是由于它不支持多维数组，也没有各种运算函数，因此也不适合做数值运算。

NumPy 的诞生弥补了这些不足，NumPy 提供了两种基本的对象：ndarray[①]和 ufunc[②]。ndarray(下文统一称之为数组)是存储单一数据类型的多维数组，而 ufunc 则是能够对数组进行处理的函数。

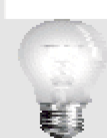
2.1 ndarray 对象

函数库的导入

本书的示例程序假设用以下推荐的方式导入 NumPy 函数库:

```
import numpy as np
```

2.1.1 创建数组



NumPy 的函数和方法都有详细的说明文档和用法示例。在 IPython 中输入函数名并添加一个“?”符号，就可以显示文档内容。例如，输入“np.array?”可以查看 array() 的说明。

首先需要创建数组才能对其进行运算和操作。可以通过给 array()函数传递 Python 的序列对象来创建数组。如果传递的是多层嵌套的序列，将创建多维数组(下例中的变量 c)：



numpy_intro.py
NumPy 的基本使用方法

① 英文全称为 n-dimensional array object。
② 英文全称为 universal function object。

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array((5, 6, 7, 8))
>>> c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
>>> b
array([5, 6, 7, 8])
>>> c
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [7, 8, 9, 10]])
```

数组的形状可以通过其 shape 属性获得，它是一个描述数组各个轴长度的元组(tuple)：

```
>>> a.shape
(4,)
>>> c.shape
(3, 4)
```

数组 a 的 shape 属性只有一个元素，因此它是一维数组。而数组 c 的 shape 属性有两个元素，因此它是二维数组，其中第 0 轴的长度为 3，第 1 轴的长度为 4。还可以通过修改数组的 shape 属性，在保持数组元素个数不变的情况下，改变数组每个轴的长度。下面的例子将数组 c 的 shape 属性改为(4,3)，注意：从(3,4)改为(4,3)并不是对数组进行转置，而只是改变每个轴的大小，数组元素在内存中的位置并没有改变。

```
>>> c.shape = 4,3
>>> c
array([[ 1,  2,  3],
       [ 4,  4,  5],
       [ 6,  7,  7],
       [ 8,  9, 10]])
```

当设置某个轴的元素个数为-1 时，将自动计算此轴的长度。由于数组 c 中有 12 个元素，因此下面的程序将数组 c 的 shape 属性改为了(2,6)：

```
>>> c.shape = 2,-1
>>> c
array([[ 1,  2,  3,  4,  4,  5],
       [ 6,  7,  7,  8,  9, 10]])
```

使用数组的 reshape()方法，可以创建指定形状的新数组，而原数组的形状保持不变：

```
>>> d = a.reshape((2,2)) # 也可以用 a.reshape(2,2)
>>> d
array([[1, 2],
```

```
        [3, 4]])
>>> a
array([1, 2, 3, 4])
```

数组 a 和 d 其实共享数据存储空间，因此修改其中任意一个数组的元素都会同时修改另外一个数组的内容：

```
>>> a[1] = 100 # 将数组 a 的第一个元素改为 100
>>> d # 注意数组 d 中的 2 也被改为了 100
array([[  1, 100],
       [  3,   4]])
```

数组的元素类型可以通过 dtype 属性获得。前面例子中，创建数组所用序列的元素都是整数，因此所创建的数组的元素类型是整型，并且是 32 bit 的长整型：

```
>>> c.dtype
dtype('int32')
```

可以通过 dtype 参数在创建数组时指定元素类型，注意 float 是 64 bit 的双精度浮点类型，而 complex 是 128 bit 的双精度复数类型：

```
>>> np.array([1, 2, 3, 4], dtype=np.float)
array([ 1.,  2.,  3.,  4.])
>>> np.array([1, 2, 3, 4], dtype=np.complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])
```

NumPy 中的数据类型都有几种字符串表示方式，字符串和类型之间的对应关系都存储在 typeDict 字典中，例如'd'、'double'、'float64'都表示双精度浮点类型：

```
>>> np.typeDict["d"]
<type 'numpy.float64'>
>>> np.typeDict["double"]
<type 'numpy.float64'>
>>> np.typeDict["float64"]
<type 'numpy.float64'>
```

完整的类型列表可以通过下面的语句得到，它将 typeDict 字典中所有的值转换为一个集合，从而去除其中的重复项：

```
>>> set(np.typeDict.values())
set([<type 'numpy.bool_'>     ,<type 'numpy.int8'>      ,<type 'numpy.int16'>
     <type 'numpy.float32'>   ,<type 'numpy.uint8'>     ,<type 'numpy.complex128'>
     <type 'numpy.unicode_'>  ,<type 'numpy.uint64'>    ,<type 'numpy.int64'>
     <type 'numpy.complex64'> ,<type 'numpy.string_'>   ,<type 'numpy.uint32'>
```

```
<type 'numpy.void'>      ,<type 'numpy.int32'>    ,<type 'numpy.float96'>
<type 'numpy.object_'>   ,<type 'numpy.uint32'>    ,<type 'numpy.int32'>
<type 'numpy.float64'>   ,<type 'numpy.complex192'>,<type 'numpy.uint16'> ])
```

前面的例子都是先创建一个 Python 的序列对象，然后通过 array()将其转换为数组，这样做显然效率不高。因此 NumPy 提供了很多专门用于创建数组的函数。下面的每个函数都有一些关键字参数，具体用法请查看函数说明。

arange()类似于内置函数 range()，通过指定开始值、终值和步长创建表示等差数列的一维数组，注意得到的结果数组不包含终值。例如下面的程序创建开始值为 0、终值为 1、步长为 0.1 的等差数组，注意终值 1 不在数组中：

```
>>> np.arange(0,1,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

linspace()通过指定开始值、终值和元素个数创建表示等差数列的一维数组，可以通过 endpoint 参数指定是否包含终值，默认值为 True，即包含终值。下面两个例子分别演示了 endpoint 为 True 和 False 时的结果，注意 endpoint 的值会改变数组的等差步长：

```
>>> np.linspace(0, 1, 10) # 步长为1/9
array([ 0.        ,  0.11111111,  0.22222222,  0.33333333,  0.44444444,
        0.55555556,  0.66666667,  0.77777778,  0.88888889,  1.        ])
>>> np.linspace(0, 1, 10, endpoint=False) # 步长为1/10
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

logspace()和 linspace()类似，不过它所创建的数组是等比数列。下面的例子产生从 10^0 到 10^2、有 5 个元素的等比数列，注意起始值 0 表示 10^0，而终值 2 表示 10^2：

```
>>> np.logspace(0, 2, 5)
array([   1.        ,    3.16227766,   10.        ,   31.6227766 ,  100.        ])
```

基数可以通过 base 参数指定，默认值为 10。下面通过将 base 参数设置为 2，并设置 endpoint 参数为 False，创建一个比例为 $2^{1/12}$ 等比数组[③]：

```
>>> np.logspace(0, 1, 12, base=2, endpoint=False)
array([ 1.        ,  1.05946309,  1.12246205,  1.18920712,  1.25992105,
        1.33483985,  1.41421356,  1.49830708,  1.58740105,  1.68179283,
        1.78179744,  1.88774863])
```

zeros()、ones()、empty()可以创建指定形状和类型的数组。其中：empty()仅仅分配数组所使用的内存，不对数组元素进行初始化操作，因此它的运行速度是最快的。下面的程序创建一个形状为(2,3)、元素类型为整数的数组：

③ 此等比数组的比值是音乐中相差半音的两个音阶之间的频率比值，因此可以用它计算一个八度中所有半音的频率。

```
>>> np.empty((2,3),np.int)     #只分配内存，不对其进行初始化
array([[  32571594,   32635312,  505219724],
       [  45001384, 1852386928,     665972]])
```

而 zeros()则将数组元素初始化为 0，ones()将数组元素初始化为 1。下面创建一个长度为 4、元素类型为浮点数的一维数组，并且元素全部初始化为 0：

```
>>> np.zeros(4, np.float)     #元素类型默认为 np.float，因此这里可以省略
array([ 0.,  0.,  0.,  0.])
```

此外，zeros_like()、ones_like()、empty_like()等函数可创建与参数数组的形状及类型相同的数组。因此，“zeros_like(a)”和“zeros(a.shape, a.dtype)”的效果相同。

使用 frombuffer()、fromstring()、fromfile()等函数可以从字节序列或文件创建数组，下面以 fromstring()为例介绍它们的用法。先创建一个包含 8 个字符的字符串 s：

```
>>> s = "abcdefgh"
```

Python 的字符串实际上是一个字节序列，每个字符占一个字节，因此如果从字符串 s 创建一个 8 bit 的整数数组，所得到的数组正好就是字符串中每个字符的 ASCII 编码：

```
>>> np.fromstring(s, dtype=np.int8)
array([ 97,  98,  99, 100, 101, 102, 103, 104], dtype=int8)
```

如果从字符串 s 创建 16 bit 的整数数组，那么两个相邻的字节就表示一个整数，把字节 98 和字节 97 当作一个 16 位的整数，它的值就是 98*256+97 = 25185。可以看出，16 bit 的整数是以低位字节在前(little-endian)的方式保存在内存中的。

```
>>> np.fromstring(s, dtype=np.int16)
array([25185, 25699, 26213, 26727], dtype=int16)
>>> 98*256+97
25185
```

如果把整个字符串转换为一个 64 bit 的双精度浮点数数组，那么它的值是：

```
>>> np.fromstring(s, dtype=np.float)
array([  8.54088322e+194])
```

显然这个结果没有什么意义，但是如果我们用 C 语言的二进制方式写了一组 double 类型的数值到某个文件中，那么就可以从此文件中读取相应的数据，并通过 fromstring()将其转换为 float64 类型的数组。或者直接使用 fromfile()从二进制文件中读取数据。

还可以先定义一个从下标计算数值的函数，然后用 fromfunction()通过此函数创建数组：

```
>>> def func(i):
```

```
...     return i%4+1
...
>>> np.fromfunction(func, (10,))
array([ 1.,  2.,  3.,  4.,  1.,  2.,  3.,  4.,  1.,  2.])
```

fromfunction()的第一个参数为计算每个数组元素的函数，第二个参数指定数组的形状。因为它支持多维数组，所以第二个参数必须是一个序列。上例中第二个参数是长度为 1 的元组(10,)，因此创建了一个有 10 个元素的一维数组。

下面的例子创建一个表示九九乘法表的二维数组，输出的数组 a 中的每个元素 a[i, j]都等于 func2(i, j)：

```
>>> def func2(i, j):
...     return (i+1) * (j+1)
...
>>> a = np.fromfunction(func2, (9,9))
>>> a
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [  2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.],
       [  3.,   6.,   9.,  12.,  15.,  18.,  21.,  24.,  27.],
       [  4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.],
       [  5.,  10.,  15.,  20.,  25.,  30.,  35.,  40.,  45.],
       [  6.,  12.,  18.,  24.,  30.,  36.,  42.,  48.,  54.],
       [  7.,  14.,  21.,  28.,  35.,  42.,  49.,  56.,  63.],
       [  8.,  16.,  24.,  32.,  40.,  48.,  56.,  64.,  72.],
       [  9.,  18.,  27.,  36.,  45.,  54.,  63.,  72.,  81.]])
```

2.1.2 存取元素



numpy_access1d.py
一维数组的元素存取

可以使用和列表相同的方式对数组的元素进行存取：

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[5]    # 用整数作为下标可以获取数组中的某个元素
5
>>> a[3:5]  # 用切片作为下标可以获取数组的一部分，包括a[3]但不包括a[5]
array([3, 4])
>>> a[:5]   # 切片中省略开始下标，表示从a[0]开始
array([0, 1, 2, 3, 4])
```

```
>>> a[:-1]  # 下标可以使用负数，表示从数组最后往前数
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> a[2:4] = 100,101    # 下标还可以用来修改元素的值
>>> a
array([  0,   1, 100, 101,   4,   5,   6,   7,   8,   9])
>>> a[1:-1:2]   # 切片中的第三个参数表示步长，2 表示隔一个元素获取一个元素
array([  1, 101,   5,   7])
>>> a[::-1] # 省略切片的开始下标和结束下标，步长为-1，表示整个数组头尾颠倒
array([  9,   8,   7,   6,   5,   4, 101, 100,   1,   0])
>>> a[5:1:-2] # 步长为负数时，开始下标必须大于结束下标
array([  5, 101])
```

和列表不同的是，通过切片获取的新数组是原始数组的一个视图。它与原始数组共享同一块数据存储空间：

```
>>> b = a[3:7] # 通过切片产生一个新的数组 b，b 和 a 共享同一块数据存储空间
>>> b
array([101,   4,   5,   6])
>>> b[2] = -10 # 将 b 的第 2 个元素修改为-10
>>> b
array([101,   4, -10,   6])
>>> a # a 的第 5 个元素也被修改为-10
array([  0,   1, 100, 101,   4, -10,   6,   7,   8,   9])
```

除了使用切片下标存取元素之外，NumPy 还提供了整数列表、整数数组和布尔数组等几种高级下标存取方法。

当使用整数列表对数组元素进行存取时，将使用列表中的每个元素作为下标。使用列表作为下标得到的数组不和原始数组共享数据：

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3, 3, 1, 8]] # 获取数组 x 中下标为 3、3、1、8 的 4 个元素，组成一个新的数组
array([7, 7, 9, 2])
>>> b = x[[3,3,-3,8]]  #下标可以是负数
>>> b[2] = 100
>>> b
array([7, 7, 100, 2])
>>> x   # 由于数组 b 和 x 不共享数据空间，因此数组 x 中的值并没有改变
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[[3,5,1]] = -1, -2, -3 # 整数序列下标也可以用来修改元素的值
>>> x
array([10, -3,  8, -1,  6, -2,  4,  3,  2])
```

当使用整数数组作为数组下标时，将得到一个形状和下标数组相同的新数组，新数组的每个元素都是用下标数组中对应位置的值作为下标从原数组获得的值。当下标数组是一维时，结果和用列表作为下标的结果相同：

```
>>> x = np.arange(10,1,-1)
>>> x[np.array([3,3,1,8])]
array([7, 7, 9, 2])
```

而当下标是多维数组时，则得到的也是多维数组。我们可以将其理解为：先将下标数组展平为一维数组，并作为下标获得一个新的一维数组，然后再将其形状修改为下标数组的形状：

```
>>> x[np.array([[3,3,1,8],[3,3,-3,8]])]
array([[7, 7, 9, 2],
       [7, 7, 4, 2]])
>>> x[[3,3,1,8,3,3,-3,8]].reshape(2,4) # 改变数组形状
array([[7, 7, 9, 2],
       [7, 7, 4, 2]])
```

当使用布尔数组 b 作为下标存取数组 x 中的元素时，将收集数组 x 中所有在数组 b 中对应下标为 True 的元素。使用布尔数组作为下标获得的数组不和原始数组共享数据内存，注意这种方式只对应于布尔数组，不能使用布尔列表。

```
>>> x = np.arange(5,0,-1)
>>> x
array([5, 4, 3, 2, 1])
>>> # 布尔数组中下标为 0、2 的元素为 True，因此获取数组 x 中下标为 0、2 的元素
>>> x[np.array([True, False, True, False, False])]
array([5, 3])
>>> # 如果是布尔列表，则把 True 当作 1，False 当作 0，按照整数序列方式获取数组 x 中的元素
>>> x[[True, False, True, False, False]]
array([4, 5, 4, 5, 5])
>>> # 布尔数组的长度不够时，不够的部分都当作 False
>>> x[np.array([True, False, True, True])]
array([5, 3, 2])
>>> # 布尔数组下标也可以用来修改元素
>>> x[np.array([True, False, True, True])] = -1, -2, -3
>>> x
array([-1,  4, -2, -3,  1])
```

布尔数组一般不是手工产生，而是使用布尔运算的 ufunc 函数产生，关于 ufunc 函数请参照 2.2 节，下面我们举一个简单的例子说明布尔数组下标的用法：

```
>>> x = np.random.rand(10) # 产生一个长度为 10、元素值为 0 到 1 的随机数组
>>> x
```

```
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.2055047 ,  0.17567449,
        0.95799412,  0.12015178,  0.7627083 ,  0.43260184,  0.91379859])
>>> x>0.5
>>> # 数组 x 中的每个元素和 0.5 进行大小比较，得到一个布尔数组，True 表示 x 中对应的值大于 0.5
array([ True, True, True, False, False, True, False, True, False, True], dtype=bool)
>>> # 使用 x>0.5 得到的布尔数组收集 x 中的元素，因此结果就是包含 x 中所有大于 0.5 的元素的数组
>>> x[x>0.5]
array([ 0.72223939,  0.921226  ,  0.7770805 ,  0.95799412,  0.7627083, 0.91379859])
```

2.1.3 多维数组

多维数组的存取和一维数组类似，因为多维数组有多个轴，因此它的下标需要用多个值表示。NumPy 采用元组(tuple)作为数组的下标，元组中的每个元素和数组的每个轴对应。图 2-1 显示了一个形状为(6,6)的数组 a，图中用不同颜色和线型标识出各个下标对应的选择区域。

numpy_access2d.py
多维数组的元素存取

图 2-1 使用数组切片语法访问多维数组中的元素

为什么使用元组作为下标

Python 的下标语法(用[]存取序列中的元素)本身并不支持多维，但是由于可以使用任何对象作为下标，因此 NumPy 使用元组作为下标存取数组中的元素，使用元组可以很方便地表示多个轴的下标。虽然在 Python 程序中，经常用圆括号将元组的元素括起来，但其实元组的语法只需要用逗号隔开元素即可，例如“x,y=y,x”就是用元组交换变量值的一个例子。因此 a[1,2]和 a[(1,2)]完全相同，都是使用元组(1,2)作为数组 a 的下标。

读者也许会对如何创建图 2-1 中的二维数组感到好奇。它实际上是一个加法表，由纵向量(0, 10, 20, 30, 40, 50)和横向量(0, 1, 2, 3, 4, 5)的元素相加而得。可以用下面的语句创建它，至于其原理，将在后面的章节进行介绍。

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1) + np.arange(0, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

图 2-1 中的下标都是含有两个元素的元组，其中第 0 个元素与数组的第 0 轴(纵轴)对应，而第 1 个元素与数组的第 1 轴(横轴)对应。下面是图中各种多维数组切片的运算结果：

```
>>> a[0,3:5]
array([3, 4])
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
>>> a[:,2]
array([ 2, 12, 22, 32, 42, 52])
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

如果下标元组中只包含整数和切片，那么得到的数组和原始数组共享数据，它是原数组的视图。下面的例子中，数组 b 是 a 的视图，它们共享数据，因此修改 b[0]时，数组 a 中对应的元素也被修改：

```
>>> b = a[0,3:5]
>>> b[0] = -b[0]
>>> a[0, 3:5]
array([-3,  4])
```

因为数组的下标是一个元组，所以我们可以将下标元组保存起来，用同一个元组存取多个数组：

```
>>> idx = slice(None, None, 2), slice(2,None)
>>> a[idx] # 和 a[::2,2:]相同
array([[ 2, -3,  4,  5],
       [22, 23, 24, 25],
```

```
       [42, 43, 44, 45]])
>>> a[idx][idx] # 和 a[::2,2:][::2,2:]相同
array([[ 4,  5],
       [44, 45]])
```

slice 对象

在[]中可以使用以冒号隔开的两个或三个整数表示切片，但是单独生成切片对象时需要使用 slice()创建。它有三个参数，分别为开始值、结束值和间隔步长，当这些值需要省略时可以使用 None。例如，a[slice(None,None,None),2]和 a[:,2]相同。

用 Python 内置的 slice()函数创建下标比较麻烦，因此 NumPy 提供了一个 s_对象来帮助我们创建数组下标：

```
>>> np.s_[::2,2:]
(slice(None, None, 2), slice(2, None, None))
```

请注意 s_实际上是一个 IndexExpression 类的对象：

```
>>> np.s_
<numpy.lib.index_tricks.IndexExpression object at 0x015093D0>
```

s_为什么不是函数

根据 Python 的语法，只有在方括号“[]”中才能使用以冒号隔开的切片语法，如果 s_是函数，那么这些切片必须使用 slice()创建。类似的对象还有 mgrid 和 ogrid 等，在后面的介绍中我们会学习它们的用法。Python 的下标语法实际上会调用__getitem__()方法，因此我们可以很容易自己实现 s_对象的功能：

```
>>> class S(object):
...     def __getitem__(self, index):
...         return index
>>> S()[::2,2:]
(slice(None, None, 2), slice(2, None, None))
```

在多维数组的下标元组中，也可以使用整数元组或列表、整数数组和布尔数组。当在下标中使用这些对象时，所获得的数据是原始数据的副本，因此修改结果数组不会改变原始数组。图 2-2 显示了如何使用各种序列下标存取多维数组。

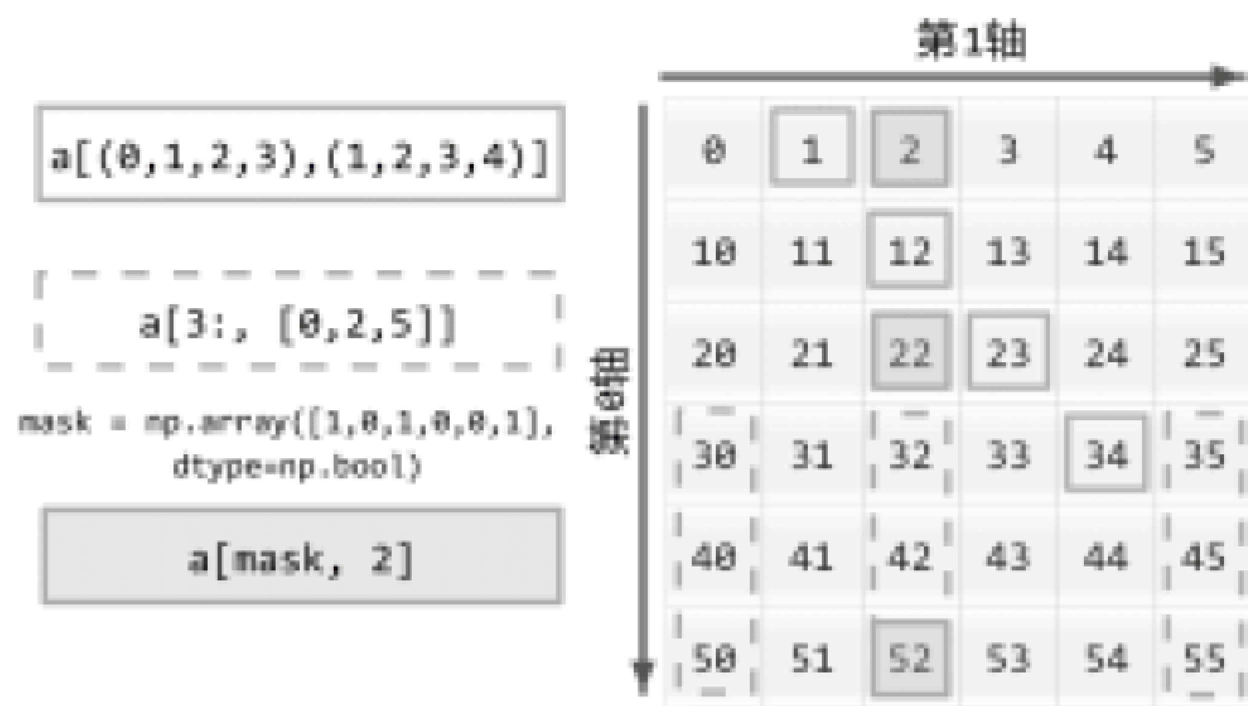


图 2-2　使用整数序列和布尔数组访问多维数组中的元素

```
>>> a[(0,1,2,3),(1,2,3,4)] ❶
array([ 1, 12, 23, 34])
>>> a[3:, [0,2,5]] ❷
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])
>>> mask = np.array([1,0,1,0,0,1], dtype=np.bool)
>>> a[mask, 2] ❸
array([ 2, 22, 52])
```

❶下标仍然是一个含有两个元素的元组，元组中的每个元素都是一个整数元组，分别对应数组的第 0 轴和第 1 轴。从两个序列的对应位置取出两个整数组成下标，于是得到的结果是：a[0,1]、a[1,2]、a[2,3]、a[3,4]。

❷第 0 轴的下标是一个切片对象，它选取第 3 行之后的所有行；第 1 轴的下标是整数列表，它选取第 0、2、5 列。

❸第 0 轴的下标是一个布尔数组，它选取第 0、2、5 行；第 1 轴的下标是一个整数，选取第 2 列。注意如果 mask 不是布尔数组而是整数数组、列表或元组，就按照❶的方式进行运算：

```
>>> mask = np.array([1,0,1,0,0,1])
>>> a[mask, 2]
array([12,  2, 12,  2,  2, 12])
>>> mask = [True,False,True,False,False,True]
>>> a[mask, 2]
array([12,  2, 12,  2,  2, 12])
```

当下标的长度小于数组的维数时，则剩余的各轴所对应的下标是“:”，即选取它们的所有数据：

```
>>> a[[1,2]]      #与 a[[1,2],:]相同
```

```
array([[10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25]])
```

当所有轴都用形状相同的整数数组作为下标时，得到的数组和下标数组的维数相同：

```
>>> x = np.array([[0,1],[2,3]])
>>> y = np.array([[-1,-2],[-3,-4]])
>>> a[x,y]
array([[ 5, 14],
       [23, 32]])
```

它的效果和下面程序的相同：

```
>>> a[(0,1,2,3),(-1,-2,-3,-4)].reshape(2,2)
array([[ 5, 14],
       [23, 32]])
```

当没有指定第 1 轴的下标时，则使用“:”作为其下标，因此得到了一个三维数组：

```
>>> a[x]
array([[[ 0,  1,  2,  3,  4,  5],
        [10, 11, 12, 13, 14, 15]],
       [[20, 21, 22, 23, 24, 25],
        [30, 31, 32, 33, 34, 35]]])
```

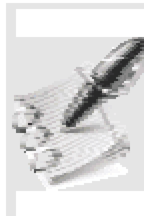
我们可以使用这种以整数数组为下标的方法来快速替换数组中的每个元素，例如有一个表示灰度图像的数组 image 以及一个调色板数组 palette，使用“palette[image]”可以得到通过调色板着色之后的彩色图像：

```
>>> palette = np.array( [ [0,0,0],     #调色板数组
...                       [255,0,0],
...                       [0,255,0],
...                       [0,0,255],
...                       [255,255,255] ] )
>>> image = np.array( [ [ 0, 1, 2, 0 ],     #图像数组
...                     [ 0, 3, 4, 0 ]  ] )
>>> palette[image]
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

2.1.4　结构数组

在 C 语言中可以通过 struct 关键字定义结构类型，结构中的字段占据连续的内存空间。类型相同的两个结构体所占用的内存大小相同，因此可以很容易定义结构数组。和 C 语言一样，在 NumPy 中也很容易对这种结构数组进行操作。只要 NumPy 中的结构定义和 C 语言中的结构定义相同，就可以很方便地读取 C 语言的结构数组的二进制数据，将其转换为 NumPy 的结构数组。

假设需要定义一个结构数组，它的每个元素都有 name、age 和 weight 字段。在 NumPy 中可以如下定义：



numpy_struct_array.py
用 NumPy 将一个结构数组写入文件中

```
persontype = np.dtype({ ❶
    'names':['name', 'age', 'weight'],
    'formats':['S32','i', 'f']}, align= True )
a = np.array([("Zhang",32,75.5),("Wang",24,65.2)], ❷
    dtype=persontype)
```

❶先创建一个 dtype 对象 persontype，它的参数是一个描述结构类型的各个字段的字典。字典有两个键：'names'和'formats'。每个键对应的值都是一个列表。'names'定义结构中每个字段的名称，而'formats'则定义每个字段的类型。这里使用类型字符串定义字段类型：

- 'S32'：长度为 32 字节的字符串类型，由于结构中每个元素的大小必须固定，因此需要指定字符串的长度。
- 'i'：32 bit 的整数类型，相当于 np.int32。
- 'f'：32 bit 的单精度浮点数类型，相当于 np.float32。

❷然后调用 array()创建数组，通过 dtype 参数指定所创建数组的元素类型为 persontype。运行上面程序之后，可以在 IPython 中执行如下语句来查看数组 a 的元素类型：

```
>>> run numpy_struct_array.py
>>> a.dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

这里我们看到了另外一种描述结构类型的方法：一个包含多个元组的列表，其中形如(字段名, 类型描述)的元组描述了结构中的每个字段。类型字符串前面的'|'、'<'、'>'等字符表示字段值的字节顺序：

- |：忽视字节顺序
- <：低位字节在前，即小端模式(little endian)
- >：高位字节在前，即大端模式(big endian)

结构数组的存取方式和一般数组相同，通过下标能够取得其中的元素，注意元素的值看上去像是元组，实际上它是一个结构：

```
>>> a[0]
('Zhang', 32, 75.5)
>>> a[0].dtype
dtype([('name', '|S32'), ('age', '<i4'), ('weight', '<f4')])
```

可以使用字段名作为下标获取对应的字段值：

```
>>> a[0]["name"]
'Zhang'
```

a[0]是一个结构元素，它和数组 a 共享内存数据，因此可以通过修改它的字段，改变原始数组中对应元素的字段：

```
>>> c = a[1]
>>> c["name"] = "Li"
>>> a[1]["name"]
"Li"
```

不但可以获得结构元素的某个字段，而且可以直接获得结构数组的字段，它返回的是原始数组的视图，因此下面的程序可以通过修改 b[0]改变 a[0]["age"]：

```
>>> b=a["age"]
>>> b
array([32, 24])
>>> b[0] = 40
>>> a[0]["age"]
40
```

通过 a.tostring()或 a.tofile()方法，可以将数组 a 以二进制的方式转换成字符串或者写入文件中：

```
>>> a.tofile("test.bin")
```

利用下面的 C 语言程序可以将“test.bin”文件中的数据读取出来。



read_struct_array.c
用 C 语言读取 NumPy 输出的结构数组文件

```
#include <stdio.h>

struct person
{
```

```
    char name[32];
    int age;
    float weight;
};
struct person p[3];
void main ()
{
    FILE *fp;
    int i;
    fp=fopen("test.bin","rb");
    fread(p, sizeof(struct person), 2, fp);
    fclose(fp);
    for(i=0;i<2;i++)
 {
        printf("%s %d %f\n", p[i].name, p[i].age, p[i].weight);
 }
}
```

内存对齐

为了内存寻址方便，C 语言的结构体会自动添加一些填充用的字节，这叫内存对齐。例如，如果把上面 C 语言所定义结构体中的字段 name[32]改为 name[30]，由于内存对齐问题，在 name 和 age 中间会填补两个字节，最终的结构体的大小不会发生改变。因此如果数组中配置的内存大小不符合 C 语言的对齐规范，将会出现数据错位。为了解决这个问题，在创建 dtype 对象时，可以传递参数 align=True，这样结构数组的内存对齐就和 C 语言的结构体一致了。

结构类型中可以包括其他的结构类型，下面的语句创建一个含有一个字段 f1 的结构，f1 的值是另外一个结构，它含有字段 f2，其类型为 16 bit 整数。

```
>>> np.dtype([('f1', [('f2', np.int16)])])
dtype([('f1', [('f2', '<i2')])])
```

当某个字段的类型为数组时，用元组的第三个参数表示其形状。在下面的结构体中，f1 字段是一个形状为(2,3)的双精度浮点数组：

```
>>> np.dtype([('f0', 'i4'), ('f1', 'f8', (2, 3))])
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

用下面的字典参数也可以定义结构类型，字典的键为结构中的字段名，值为字段的类型描述，但是由于字典的键是没有顺序的，因此字段的顺序需要在类型描述中给出。类型描述是一个元组，它的第二个值给出字段的以字节为单位的偏移量，例如下例中的 age 字段的偏移量为

25 个字节：

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', '|S25'), ('age', '|u1')])
```

使用字段的地址偏移量可以定义内存地址不连续的字段，在读取复杂的 C 语言结构体中的部分字段时这种方式十分有效。

2.1.5 内存结构

下面让我们看看数组对象是如何在内存中存储的。如图 2-3 所示，数组的描述信息保存在一个数据结构中，这个结构引用两个对象：用于保存数据的存储区域和用于描述元素类型的 dtype 对象。

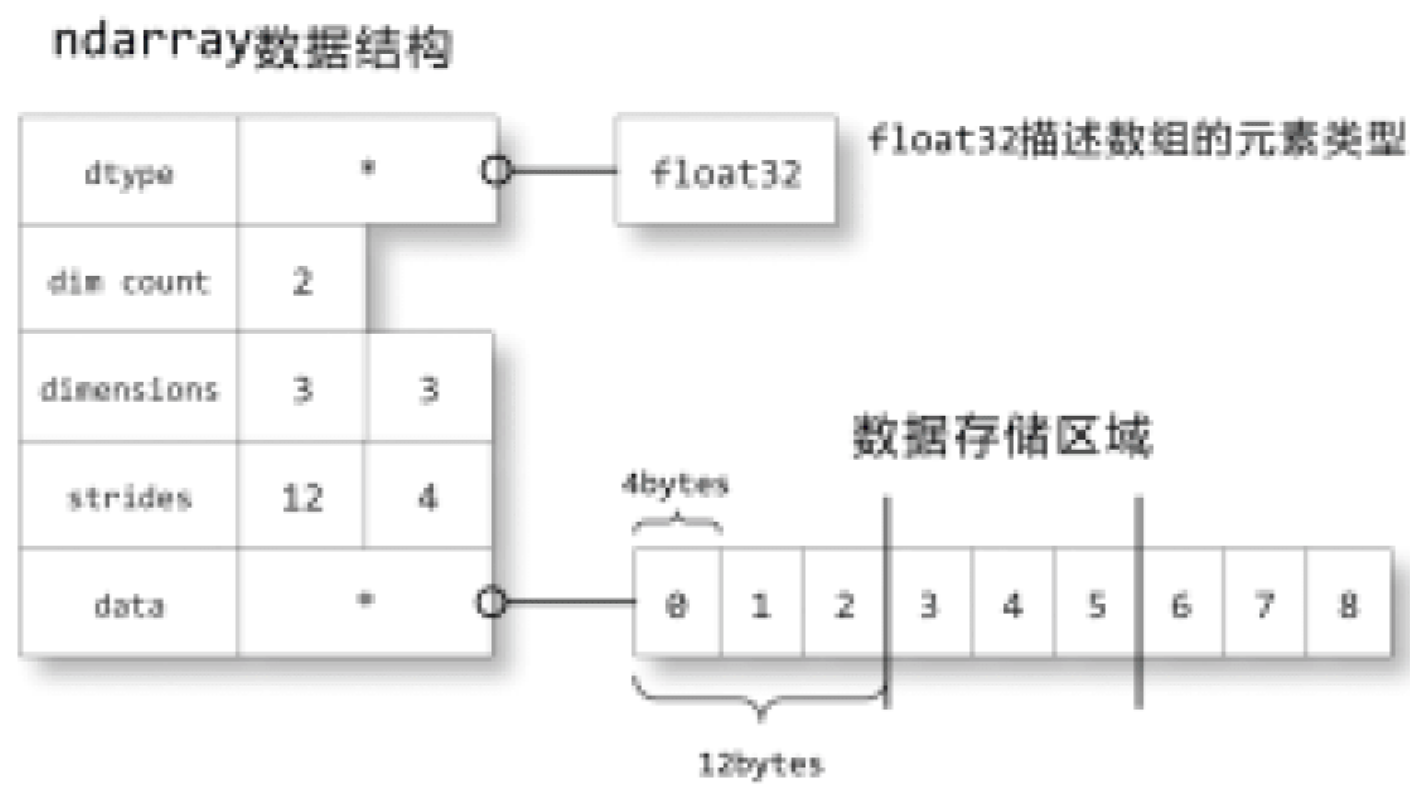


图 2-3 ndarray 数组对象在内存中的存储方式

数据存储区域保存着数组中所有元素的二进制数据，dtype 对象则知道如何将元素的二进制数据转换为可用的值。数组的维数和形状等信息都保存在 ndarray 数组对象的数据结构中。图 2-3 中显示的是下面的数组 a 的内存结构：

```
>>> a = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32)
```

数组对象使用其 strides 属性保存每个轴上相邻两个元素的地址差，即当某个轴的下标增加 1 时，数据存储区域指针所增加的字节数。例如图 2-3 中的 strides 为(12,4)，即第 0 轴的下标增加 1 时，数据的地址增加 12 个字节。也就是 a[1,0]的地址比 a[0,0]的地址大 12，正好是 3 个单精度浮点数的总字节数。第 1 轴下标增加 1 时，数据的地址增加 4 个字节，正好是一个单精度浮点数的字节数。

如果 strides 属性中的数值正好和对应轴所占据的字节数相同，那么数据在内存中是连续存储的。通过切片下标得到的新数组是原始数组的视图，即它和原始数组共享数据存储区域，但是新数组的 strides 属性和 data 属性会发生变化：

```
>>> b = a[::2,::2]
>>> b
array([[ 0.,  2.],
       [ 6.,  8.]], dtype=float32)
>>> b.strides
(24, 8)
```

由于数组 b 和数组 a 共享数据存储区域，而数组 b 中的第 0 轴和第 1 轴的元素都是从数组 a 中隔一个元素取一个，因此数组 b 的 strides 变成了(24,8)，正好都是数组 a 的两倍。对照前面的图 2-3 很容易看出：数据 0 和 2 的地址相差 8 个字节，而数据 0 和 6 的地址相差 24 个字节。

元素在数据存储区域的排列格式有两种：C 语言格式和 Fortan 语言格式。在 C 语言中，多维数组的第 0 轴是最上位的，即第 0 轴的下标增加 1 时，元素的地址增加的字节数最多。而 Fortan 语言中的多维数组的第 0 轴是最下位的，即第 0 轴的下标增加 1 时，地址只增加一个元素的字节数。在 NumPy 中，默认以 C 语言格式存储数据，如果希望改为 Fortan 格式，只需要在创建数组时，设置 order 参数为"F"即可：

```
>>> c = np.array([[0,1,2],[3,4,5],[6,7,8]], dtype=np.float32, order="F")
>>> c.strides
(4, 12)
```

了解了数组的内存结构后，就可以解释使用数组下标获取数据时的复制和引用问题：

- 当下标使用整数和切片时，获取的数据在数据存储区域中是等间隔分布的。由于只需要修改图 2-3 所示数据结构中的 dim count、dimensions、stride 等属性以及指向数据存储区域的指针 data，就能实现整数和切片下标，因此新数组和原始数组能够共享数据存储区域。
- 当使用整数序列、整数数组和布尔数组时，不能保证获取的数据在数据存储区域是等间隔的，因此无法和原始数组共享数据，只能对数据进行复制。

数组的 flags 属性描述了数据存储区域的一些属性，直接查看 flags 属性将输出各个标识的值：

```
>>> a.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

也可以单独获得其中的某个标识的值：

```
>>> a.flags.c_contiguous
True
```

几个比较重要的标识的含义如下：

- C_CONTIGUOUS：数据存储区域是否是 C 语言格式的连续区域。
- F_CONTIGUOUS：数据存储区域是否是 Fortan 语言格式的连续区域。
- OWNDATA：数组是否拥有此数据存储区域，当数组是其他数组的视图时，它不拥有数据存储区域。

由于数组 a 是通过 array()直接创建的，因此它的数据存储区域是 C 语言格式的连续区域，并且它拥有数据存储区域。下面我们看看数组 a 的转置数组的标识：

```
>>> a.T.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
```

数组的转置可以通过其 T 属性获得，转置数组将其数据存储区域看作 Fortan 语言格式的连续区域，并且它不拥有数据存储区域。

```
>>> b.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
```

由于数组 b 是数组 a 的一个视图，因此它既不拥有数据存储区域，它的数据也不是连续存储的。通过视图数组的 base 属性可以获得保存数据的原始数组：

```
>>> id(b.base)
34064760
>>> id(a)
34064760
```

除了使用切片从同一块数据区域创建不同的 shape 和 strides 的数组对象之外，我们还可以直接设置这些属性，从而得到用切片实现不了的效果，例如：

```
>>> from numpy.lib.stride_tricks import as_strided
>>> a = np.arange(6)
>>> b = as_strided(a, shape=(4, 3), strides=(4, 4))
>>> b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
```



使用 as_strided()时 NumPy 不会进行内存越界检查，因此如果 shape 和 strides 设置不当，就可能会引起意想不到的错误。

在上面这个例子中，我们从 NumPy 的辅助模块中载入了一个 as_strided()函数，并使用它从一个长度为 6 的一维数组 a，创建了一个 shape 为(4,3)的二维数组 b。由于通过 strides 参数直接指定了数组 b 的 strides 属性，因此不仅数组 b 和数组 a 共享数据区域，而且数组 b 中前后两行有两个元素是重合的。例如下面修改 a[2]的值，数组 b 中前三行中对应的元素也会发生改变：

```
>>> a[2] = 20
>>> b
array([[ 0,  1, 20],
       [ 1, 20,  3],
       [20,  3,  4],
       [ 3,  4,  5]])
```

在对数据进行处理时，可能经常需要对数据进行分块处理，而且为了保持平滑，每块数据之间需要有一定的重叠部分。这时可以使用上面介绍的方法对数据进行带重叠的分块。

2.2　ufunc 运算

ufunc 是 universal function 的缩写，它是一种能对数组中每个元素进行操作的函数。NumPy 内置的许多 ufunc 函数都是在 C 语言级别实现的，因此它们的计算速度非常快。让我们先看一个例子：

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> y
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44921271e-16])
```

先用 linspace()产生一个从 0 到 2π 的等差数组，然后将其传递给 np.sin()函数计算每个元素的正弦值。由于 np.sin()是一个 ufunc 函数，因此在其内部对数组 x 的每个元素进行循环，分别计算它们的正弦值，并返回一个保存各个计算结果的数组。运算之后数组 x 中的值并没有改变，而是新创建了一个数组来保存结果。也可以通过 out 参数指定计算结果的保存位置。因此如果希望直接在数组 x 中保存结果，可以将它传递给 out 参数：

```
>>> t = np.sin(x,out=x)
>>> x
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
```

```
        -2.44921271e-16])
>>> id(t) == id(x)
True
```

ufunc 函数的返回值仍然是计算的结果，只不过它就是数组 x，因此两个数组的 id 是相同的。下面比较 np.sin 和 Python 标准库中 math.sin 的计算速度：



numpy_speed_test.py
NumPy 计算和 Python 标准库的计算速度比较

```
import time
import math
import numpy as np

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = math.sin(t)
print "math.sin:", time.clock() - start

x = [i * 0.001 for i in xrange(1000000)]
x = np.array(x)
start = time.clock()
np.sin(x,x)
print "numpy.sin:", time.clock() - start

x = [i * 0.001 for i in xrange(1000000)]
start = time.clock()
for i, t in enumerate(x):
    x[i] = np.sin(t)
print "numpy.sin loop:", time.clock() - start
```

程序的输出为：

```
math.sin: 0.779217749742
numpy.sin: 0.0772958574344
numpy.sin loop: 5.25540843878
```

可以看出 np.sin 比 math.sin 快 10 倍多，这得益于 np.sin 在 C 语言级别的循环计算。

列表推导式比循环更快

事实上，标准 Python 中有比 for 循环更快的方案——使用列表推导式。但是列表推导式将产生一个新的列表，而不是直接修改原来列表中的元素。下面的语句执行时，将计算出一

个新的列表来保存每个正弦值：

```
>>> x = [math.sin(t) for t in x]
```

np.sin 同样也支持计算单个数值的正弦值。不过值得注意的是，对单个数值的计算 math.sin 要比 np.sin 快很多。在 Python 级别进行循环时，np.sin 的计算速度只有 math.sin 的 1/6。这是因为 np.sin 为了同时支持数组和单个数值的计算，其 C 语言的内部实现要比 math.sin 复杂得多。此外，对于单个数值的计算，np.sin 的返回值类型和 math.sin 的不同，math.sin 返回的是 Python 的标准 float 类型，而 np.sin 返回的是 float64 类型：

```
>>> type(math.sin(0.5))
<type 'float'>
>>> type(np.sin(0.5))
<type 'numpy.float64'>
```

通过下标获取的数组元素的类型为 NumPy 中定义的类型。将其转换为 Python 的标准类型还需要花费额外的时间。为了解决这个问题，数组提供了 item()方法，用来获取数组中的单个元素，并直接返回标准的 Python 数值类型：

```
>>> a = np.arange(6.0).reshape(2,3)
>>> a.item(1,2) # 和 a[1,2]类似
5.0
>>> type(a.item(1,2)) # item()返回的是 Python 的标准 float 类型
<type 'float'>
>>> type(a[1,2]) # 下标方式返回的是 NumPy 的 float64 类型
<type 'numpy.float64'>
```

通过上面的例子我们了解了如何最有效率地使用 math 库和 NumPy 中的数学函数。由于它们各有优缺点，因此在导入时不建议使用“import *”全部载入，而是应该使用“import numpy as np”载入，这样可以根据需要选择合适的函数。

2.2.1 四则运算

NumPy 提供许多 ufunc 函数，例如计算两个数组之和的 add()函数：

```
>>> a = np.arange(0,4)
>>> a
array([0, 1, 2, 3])
>>> b = np.arange(1,5)
>>> b
array([1, 2, 3, 4])
>>> np.add(a,b)
array([1, 3, 5, 7])
```

```
>>> np.add(a,b,a)
array([1, 3, 5, 7])
>>> a
array([1, 3, 5, 7])
```

add()返回一个数组，数组的每个元素都是两个参数数组中对应元素之和。如果没有指定 out 参数，那么将创建一个新的数组来保存计算结果。如果指定了第三个参数 out，就不产生新的数组，而是直接将结果保存到指定的数组中。

NumPy 为数组定义了各种数学运算操作符，因此计算两个数组相加可以简单地写为 a+b，而 np.add(a,b,a)则可以用 a+=b 表示。表 2-1 列出了数组的运算符及其对应的 ufunc 函数，注意除号"/"的意义根据是否激活__future__.division 会有所不同。

表 2-1　数组的运算符及其对应的 ufunc 函数

运 算 符	对应的 ufunc 函数
y = x1 + x2	add(x1, x2 [, y])
y = x1 - x2	subtract(x1, x2 [, y])
y = x1 * x2	multiply (x1, x2 [, y])
y = x1 / x2	divide (x1, x2 [, y])，如果两个数组的元素为整数，那么用整数除法
y = x1 / x2	true_divide (x1, x2 [, y])，总是返回精确的商
y = x1 // x2	floor_divide (x1, x2 [, y])，总是对返回值取整
y = -x	negative(x [,y])
y = x1**x2	power(x1, x2 [, y])
y = x1 % x2	remainder(x1, x2 [, y]),或 mod(x1, x2, [, y])

数组对象支持操作符，极大地简化了表达式的编写。不过要注意：如果表达式很复杂，并且要运算的数组很大，将会因为产生大量的中间结果而降低程序的运算效率。例如，假设对 a、b、c 三个数组采用表达式“x=a*b+c”进行计算，那么这个表达式相当于：

```
t = a * b
x = t + c
del t
```

也就是说，需要产生一个临时数组 t 来保存乘法的运算结果，然后再产生最后的结果数组 x。可以将表达式分解为下面的两行语句，以减少一次内存分配：

```
x = a*b
x += c
```

2.2.2 比较和布尔运算

使用“==”、“>”等比较运算符对两个数组进行比较，将返回一个布尔数组，它的每个元素值都是两个数组对应元素的比较结果。例如：

```
>>> np.array([1,2,3]) < np.array([3,2,1])
array([ True, False, False], dtype=bool)
```

每个比较运算符都与一个 ufunc 函数对应，下面是比较运算符与其 ufunc 函数的对照表：

表 2-2　比较运算符及其对应的 ufunc 函数

比较运算符	ufunc 函数
y = x1 == x2	equal(x1, x2 [, y])
y = x1 != x2	not_equal(x1, x2 [, y])
y = x1 < x2	less(x1, x2, [, y])
y = x1 <= x2	less_equal(x1, x2, [, y])
y = x1 > x2	greater(x1, x2, [, y])
y = x1 >= x2	greater_equal(x1, x2, [, y])

由于 Python 中的布尔运算使用 and、or 和 not 等关键字，它们无法被重载，因此数组的布尔运算只能通过相应的 ufunc 函数进行。这些函数名都以“logical_”开头，在 IPython 中使用自动补全功能可以很容易地找到它们：

```
>>> np.logical # 按 Tab 键进行自动补全
np.logical_and np.logical_not np.logical_or  np.logical_xor
```

下面是一个使用 logical_or()进行或运算的例子：

```
>>> a = np.arange(5)
>>> b = np.arange(4,-1,-1)
>>> a == b
array([False, False,  True, False, False], dtype=bool)
>>> a > b
array([False, False, False,  True,  True], dtype=bool)
>>> np.logical_or(a==b, a>b) # 和 a>=b 相同
array([False, False,  True,  True,  True], dtype=bool)
```

对两个布尔数组使用 and、or 和 not 等进行布尔运算，将抛出 ValueError 异常。因为布尔数组中有 True 也有 False，所以 NumPy 无法确定用户的运算目的：

```
>>> a==b and a>b
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

错误信息告诉我们可以使用数组的 any()或 all()方法[④]。只要数组中有一个值为 True，any()就返回 True；而只有当数组的全部元素都为 True 时，all()才返回 True。

```
>>> np.any(a==b)
True
>>> np.any(a==b) and np.any(a>b)
True
```

以“bitwise_”开头的函数是比特运算函数，包括 bitwise_and、bitwise_not、bitwise_or 和 bitwise_xor 等。也可以使用"&"、"~"、"|"和"^"等操作符进行计算。

对于布尔数组来说，位运算和布尔运算的结果相同。但在使用时要注意，位运算符的优先级比比较运算符高，因此需要使用括号提高比较运算的运算优先级。例如：

```
>>> (a==b) | (a>b)
array([False, False,  True,  True,  True], dtype=bool)
```

整数数组的位运算和 C 语言的位运算相同，在使用时要注意元素类型的符号，例如下面的 arange()所创建的数组的元素类型为 32 位符号整数，因此对正数按位取反将得到负数。以整数 0 为例，按位取反的结果是 0xFFFFFFFF，在 32 位符号整数中，这个值表示-1。

```
>>> ~np.arange(5)
array([-1, -2, -3, -4, -5])
```

而如果对 8 位无符号整数数组进行位取反运算：

```
>>> ~np.arange(5, dtype=np.uint8)
array([255, 254, 253, 252, 251], dtype=uint8)
```

同样的整数 0，按位取反的结果是 0xFF，当它是 8 位无符号整数时，它的值是 255。

2.2.3 自定义 ufunc 函数

通过 NumPy 提供的标准 ufunc 函数，可以组合出复杂的表达式，在 C 语言级别对数组的每个元素进行计算。但有时这种表达式不易编写，而对每个元素进行计算的程序却很容易用 Python 实现，这时可以用 frompyfunc()将一个计算单个元素的函数转换成 ufunc 函数。这样就可以方便地用所产生的 ufunc 函数对数组进行计算了。

例如，我们可以用一个分段函数描述三角波，三角波的样子如图 2-4 所示，它分为三段：上升段、下降段和平坦段。

④ 在 NumPy 中同时也定义了 any()和 all()函数。

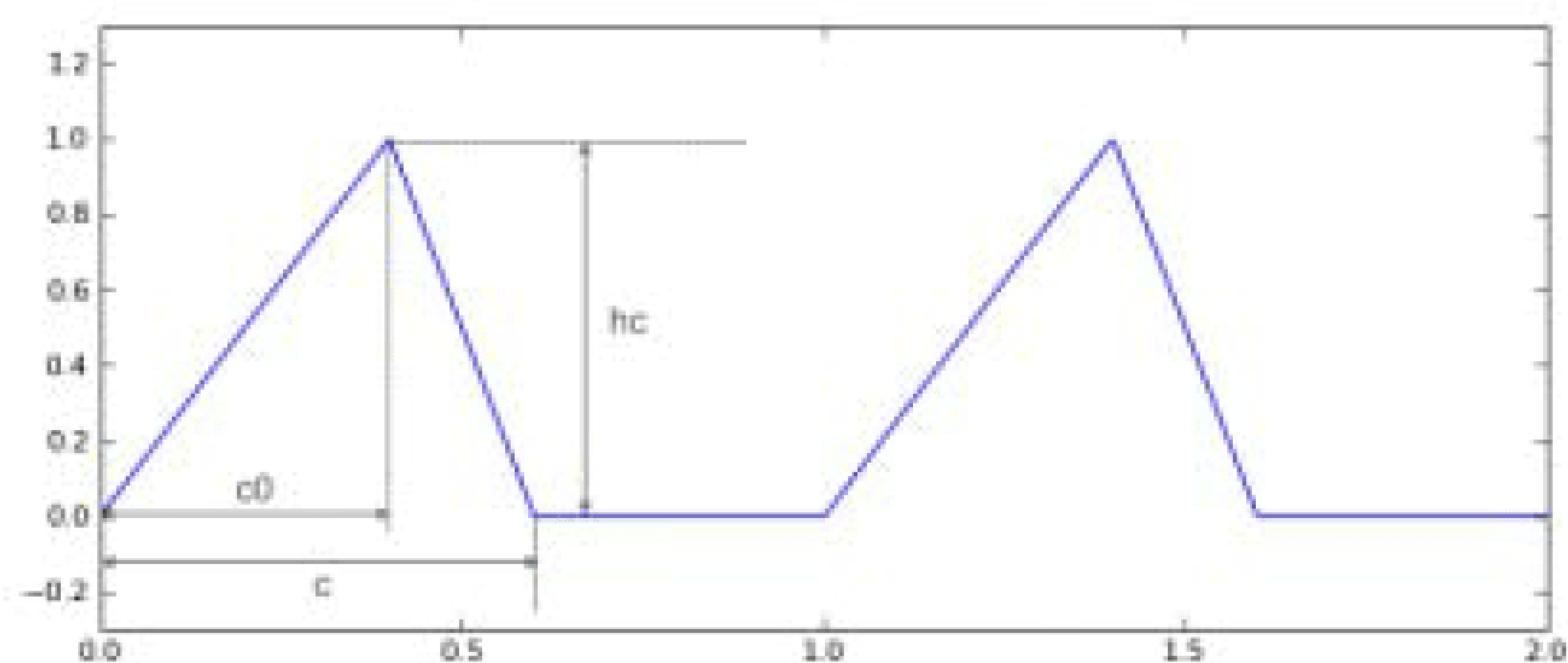


图 2-4　三角波可以用分段函数进行计算



numpy_frompyfunc.py
用 frompyfunc()计算三角波的波形数组

根据图 2-4，我们很容易写出计算三角波上某点 Y 坐标的函数。显然 triangle_wave()只能计算单个数值，不能对数组直接进行处理。

```
def triangle_wave(x, c, c0, hc):
    x = x - int(x) # 三角波的周期为 1，因此只取 x 坐标的小数部分进行计算
    if x >= c: r = 0.0
    elif x < c0: r = x / c0 * hc
    else: r = (c-x) / (c-c0) * hc
    return r
```

可以用下面的程序先使用列表推导式计算出一个列表，然后用 array()将列表转换为数组。这种做法每次都需要使用列表推导式语法调用函数，对于多维数组是很麻烦的。

```
x = np.linspace(0, 2, 1000)
y1 = np.array([triangle_wave(t, 0.6, 0.4, 1.0) for t in x])
```

通过 frompyfunc()可以将计算单个值的函数转换为一个能对数组中每个元素进行计算的 ufunc 函数。frompyfunc()的调用格式为：

```
frompyfunc(func, nin, nout)
```

其中，func 是计算单个元素的函数，nin 是 func 输入参数的个数，nout 是 func 返回值的个数。下面的程序使用 frompyfunc()将 triangle_wave()转换为一个 ufunc 函数对象 triangle_ufunc1：

```
triangle_ufunc1 = np.frompyfunc(triangle_wave, 4, 1)
y2 = triangle_ufunc1(x, 0.6, 0.4, 1.0)
```

值得注意的是，triangle_ufunc1()所返回数组的元素类型是 object，因此还需要再调用数组的 astype()方法以将其转换为双精度浮点数组：

```
>>> run numpy_frompyfunc.py # 在 IPython 中运行计算三角波的程序
>>> y2.dtype
dtype('object')
>>> y2 = y2.astype(np.float)
>>> y2.dtype
dtype('float64')
```

使用 vectorize()可以实现和 frompyfunc()类似的功能，但它可以通过 otypes 参数指定返回数组的元素类型。otypes 参数可以是一个表示元素类型的字符串，也可以是一个类型列表，使用列表可以描述多个返回数组的元素类型。下面的程序使用 vectorize()计算三角波：

```
triangle_ufunc2 = np.vectorize(triangle_wave, otypes=[np.float])
y3 = triangle_ufunc2(x, 0.6, 0.4, 1.0)
```

最后我们验证一下结果：

```
>>> np.all(y1==y2)
True
>>> np.all(y2==y3)
True
```

2.2.4 广播

当使用 ufunc 函数对两个数组进行计算时，ufunc 函数会对这两个数组的对应元素进行计算，因此要求这两个数组的形状相同。如果形状不同，会进行如下的广播(broadcasting)处理：

(1) 让所有输入数组都向其中维数最多的数组看齐，shape 属性中不足的部分都通过在前面加 1 补齐。

(2) 输出数组的 shape 属性是输入数组的 shape 属性在各个轴上的最大值。

(3) 如果输入数组的某个轴长度为 1 或与输出数组对应轴的长度相同，这个数组就能够用来计算，否则出错。

(4) 当输入数组的某个轴长度为 1 时，沿着此轴运算时都用此轴上的第一组值。

上述 4 条规则理解起来可能比较费劲，下面让我们看一个实际的例子。

先创建一个二维数组 a，其形状为(6,1)：

```
>>> a = np.arange(0, 60, 10).reshape(-1, 1)
>>> a
array([[ 0], [10], [20], [30], [40], [50]])
>>> a.shape
(6, 1)
```

再创建一维数组 b，其形状为(5,)：

```
>>> b = np.arange(0, 5)
>>> b
array([0, 1, 2, 3, 4])
>>> b.shape
(5,)
```

计算数组 a 和 b 的和，得到一个加法表，它相当于计算两个数组中所有元素组的和，得到一个形状为(6,5)的数组：

```
>>> c = a + b
>>> c
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44],
       [50, 51, 52, 53, 54]])
>>> c.shape
(6, 5)
```

由于数组 a 和 b 的维数不同，根据规则(1)，需要让数组 b 的 shape 属性向数组 a 对齐，于是将数组 b 的 shape 属性前面加 1，补齐为(1,5)。相当于做了如下计算：

```
>>> b.shape=1,5
>>> b
array([[0, 1, 2, 3, 4]])
```

这样一来，做加法运算的两个输入数组的 shape 属性分别为(6,1)和(1,5)，根据规则(2)，输出数组各个轴的长度为输入数组各个轴长度的最大值，可知输出数组的 shape 属性为(6,5)。

由于数组 b 第 0 轴的长度为 1，而数组 a 第 0 轴的长度为 6，因此为了让它们在第 0 轴上能够相加，需要将数组 b 第 0 轴的长度扩展为 6，这相当于：

```
>>> b = b.repeat(6,axis=0)
>>> b
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

由于数组 a 第 1 轴的长度为 1，而数组 b 第 1 轴的长度为 5，因此为了让它们在第 1 轴上能够相加，需要将数组 a 第 1 轴的长度扩展为 5，这相当于：

```
>>> a = a.repeat(5, axis=1)
>>> a
array([[ 0,  0,  0,  0,  0],
       [10, 10, 10, 10, 10],
       [20, 20, 20, 20, 20],
       [30, 30, 30, 30, 30],
       [40, 40, 40, 40, 40],
       [50, 50, 50, 50, 50]])
```

经过上述处理之后，数组 a 和 b 就可以按对应元素进行相加运算了。

当然，在执行“a+b”运算时，NumPy 内部并不会真正将长度为 1 的轴用 repeat()进行扩展，这样太浪费空间了。

由于这种广播计算很常用，因此 NumPy 提供了快速产生能进行广播运算的数组的 ogrid 对象。

```
>>> x,y = np.ogrid[:5,:5]
>>> x
array([[0],[1],[2],[3],[4]])
>>> y
array([[0, 1, 2, 3, 4]])
```



mgrid 对象的用法和 ogrid 对象类似，但是它所返回的是进行广播之后的数组。请读者运行“np.mgrid[:5,:5]”试试看。

ogrid 是一个很有趣的对象，它和多维数组一样，用切片元组作为下标，返回的是一组可以用来广播计算的数组。其切片下标有两种形式：

- 开始值:结束值:步长，和“np.arange(开始值, 结束值, 步长)”类似。
- 开始值:结束值:长度 j，当第三个参数为虚数时，它表示所返回数组的长度，其和“np.linspace(开始值, 结束值, 长度)”类似。

```
>>> x, y = np.ogrid[:1:4j, :1:3j]
>>> x
array([[ 0.        ],
       [ 0.33333333],
       [ 0.66666667],
       [ 1.        ]])
>>> y
array([[ 0. ,  0.5,  1. ]])
```

利用 ogrid 的返回值，可以很容易计算出二元函数在等间距网格上的值。下面是绘制三维曲面 $f(x,y)=xe^{x^2-y^2}$ 的程序：



numpy_ogrid_mlab.py
用 ogird 产生二维坐标网格，计算三维空间的曲面

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2)

pl = mlab.surf(x, y, z, warp_scale="auto")
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(pl)
mlab.show()
```

此程序使用 Mayavi 的 mlab 模块快速绘制如图 2-5 所示的 3D 曲面(见封二彩插)，关于 Mayavi 的相关内容将在随后的章节中进行介绍。

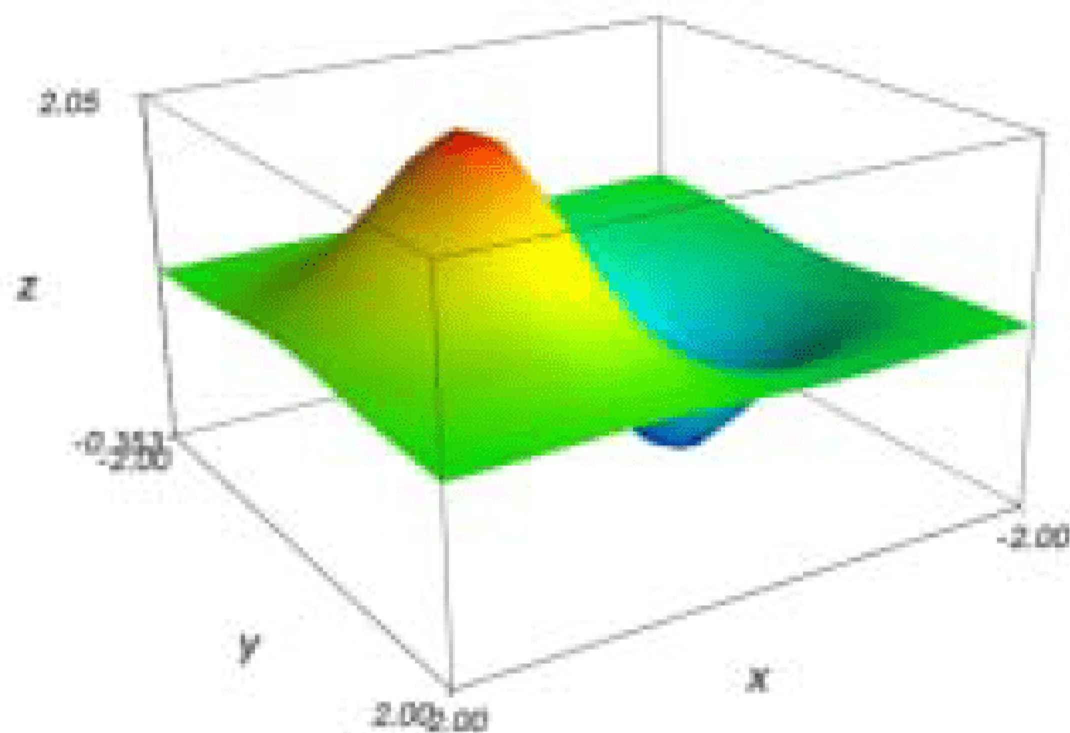


图 2-5　使用 ogrid 创建的三维曲面

如果已经有了多个表示在不同轴上取值的一维数组，并且想计算出由它们所构成网格上的每点的函数值，那么可以使用 ix_()：

```
>>> x = np.array([0, 1, 4, 10])
>>> y = np.array([2, 3, 8])
>>> gy, gx = np.ix_(y, x)
>>> gx
array([[ 0,  1,  4, 10]])
>>> gy
array([[2], [3], [8]])
```

```
>>> gx+gy # 广播运算
array([[ 2,  3,  6, 12],
       [ 3,  4,  7, 13],
       [ 8,  9, 12, 18]])
```

在上面的例子中，通过 ix_()将数组 x 和 y 转换成能进行广播运算的二维数组。注意数组 y 对应广播运算结果中的第 0 轴，而数组 x 与第 1 轴对应。

2.2.5 ufunc 函数的方法

ufunc 函数对象本身还有一些方法，这些方法只对两个输入、一个输出的 ufunc 对象有效，其他的 ufunc 对象调用这些方法时会抛出 ValueError 异常。

reduce()方法和 Python 的 reduce()函数类似，它沿着 axis 参数指定的轴对数组进行操作，相当于将<op>运算符插入到沿 axis 轴的所有元素之间。

```
<op>.reduce (array, axis=0, dtype=None)
```

例如：

```
>>> np.add.reduce([1,2,3]) # 1 + 2 + 3
6
>>> np.add.reduce([[1,2,3],[4,5,6]], axis=1) # (1+2+3),(4+5+6)
array([ 6, 15])
```

accumulate()和 reduce()类似，只是它返回的数组和输入数组的形状相同，保存所有的中间计算结果：

```
>>> np.add.accumulate([1,2,3])
array([1, 3, 6])
>>> np.add.accumulate([[1,2,3],[4,5,6]], axis=1)
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

reduceat()计算多组 reduce()的结果，通过 indices 参数指定一系列的起始和终止位置。它的计算有些特别，让我们通过例子详细解释一下：

```
>>> a = np.array([1,2,3,4])
>>> result = np.add.reduceat(a,indices=[0,1,0,2,0,3,0])
>>> result
array([ 1,  2,  3,  3,  6,  4, 10])
```

对于 indices 参数中的每个元素都会计算出一个值，因此最终的计算结果和 indices 参数的长度相同。结果数组 result 中除最后一个元素之外，都按照如下计算得出：

```
if indices[i] < indices[i+1]:
    result[i] = <op>.reduce(a[indices[i]:indices[i+1]])
else:
    result[i] = a[indices[i]]
```

最后一个元素则按照如下计算得出：

```
<op>.reduce(a[indices[-1]:])
```

因此上面例子中，数组 result 的每个元素都按照如下计算得出：

```
1 : a[0] -> 1
2 : a[1] -> 2
3 : a[0] + a[1] -> 1 + 2
3 : a[2] -> 3
6 : a[0] + a[1] + a[2] ->  1 + 2 + 3 = 6
4 : a[3] -> 4
10: a[0] + a[1] + a[2] + a[4] -> 1 + 2 + 3 + 4 = 10
```

可以看出：result[::2]和 a 相等，而 result[1::2]和 np.add.accumulate(a)相等。使用 reduceat()可以对数组中的多个片段同时进行 reduce 运算。

ufunc 函数对象的 outer()方法对其两个参数数组中每两对元素的组合进行运算。如果数组 a 的维数为 M，数组 b 的维数为 N，那么使用 ufunc 函数 op 的 outer()方法对数组 a、b 计算后，生成的数组 c 的维数为 M+N。数组 c 的形状是数组 a、b 形状的结合。例如数组 a 的形状为(2,3)，数组 b 的形状为(4,5)，则数组 c 的形状为(2,3,4,5)。

让我们看一个例子：

```
>>> np.multiply.outer([1,2,3,4,5],[2,3,4])
array([[ 2,  3,  4],
       [ 4,  6,  8],
       [ 6,  9, 12],
       [ 8, 12, 16],
       [10, 15, 20]])
```

可以看出，通过 outer()计算得到的结果是如下的乘法表：

```
*| 2  3  4
------------
1| 2  3  4
2| 4  6  8
3| 6  9 12
4| 8 12 16
5|10 15 20
```

如果将这两个数组按照等同程序一步一步地进行计算，就会发现乘法表最终是通过广播的方式计算出来的。

2.3 多维数组的下标存取

在前面的介绍中，我们通过一些实例介绍了如何对多维数组进行下标访问。实际上，NumPy 提供的下标功能十分强大，在掌握了“广播”相关的知识之后，让我们再回过头来系统地学习数组的下标规则。

2.3.1 下标对象

首先，多维数组的下标应该是一个长度上与数组的维数相同的元组。如果下标元组的长度比数组的维数大，就会出错。如果小，就需要在下标元组的后面补“:”，使得它的长度与数组维数相同。

如果下标对象不是元组，NumPy 会首先把它转换为元组。这种转换可能会和用户所希望的不一致，因此为了避免出现问题，请显式地使用元组作为下标。例如，数组 a 是一个三维数组，下面分别使用一个二维列表 lidx 和一个二维数组 aidx 作为下标，得到的结果是不一样的。

```
>>> a = np.arange(3*4*5).reshape(3,4,5)
>>> lidx = [[0],[1],[2]]
>>> aidx = np.array(lidx)
>>> a[lidx]
array([7])
>>> a[aidx]
array([[[[ 0,  1,  2,  3,  4],
         [ 5,  6,  7,  8,  9],
[[省略]]
```

这是因为 NumPy 将列表 lidx 转换成了([0],[1],[2])，而将数组 aidx 转换成了[aidx,:,:]：

```
>>> a[tuple(lidx)]
array([7])
>>> a[aidx,:,:]
array([[[[ 0,  1,  2,  3,  4],
         [ 5,  6,  7,  8,  9],
[[省略]]
```

经过各种转换和添加“:”之后，得到了一个标准的下标元组。它的各个元素有如下几种类型：切片、整数、整数数组和布尔数组。如果元素不是这些类型，如列表或元组，就将其转换成整数数组。

如果下标元组的所有元素都是切片和整数，那么用它作为下标得到的是原始数组的一个视图，即它和原始数组共享数据存储空间。

2.3.2 整数数组作为下标

下面让我们看看下标元组中的元素由切片和整数数组构成的情况。假设整数数组有 N_c 个，切片有 N_s 个。N_c+N_s 为数组的维数 D。

首先这 N_c 个整数数组必须满足广播条件，假设它们进行广播之后的维数为 M，形状为 $(d_0,d_1,\ldots,d_{m-1})$。

如果 N_s 为 0，即没有切片元素，那么下标得到的结果数组 result 的形状和整数数组广播之后的形状相同。它的每个元素值可按照下面的公式得出：

$$result[i_0,i_1,\ldots,i_{m-1}]=X[ind_0[i_0,i_1,\ldots,i_{m-1}],\ldots,ind_{N_c-1}[i_0,i_1,\ldots,i_{m-1}]]$$

其中 ind_0 到 ind_{N_c-1} 为进行广播之后的整数数组。让我们通过一个例子加深对此公式的理解：

```
>>> i0 = np.array([[1,2,1],[0,1,0]])
>>> i1 = np.array([[[0]],[[1]]])
>>> i2 = np.array([[[2,3,2]]])
>>> b = a[i0, i1, i2]
>>> b
array([[[22, 43, 22],
        [ 2, 23,  2]],
       [[27, 48, 27],
        [ 7, 28,  7]]])
```

首先，i0、i1、i2 三个整数数组的 shape 属性分别为(2,3)、(2,1,1)、(1,1,3)，根据广播规则，先在长度不足 3 的 shape 属性前面补 1，使它们的维数相同，广播之后的 shape 属性为各个轴的最大值：

```
(1, 2, 3)
(2, 1, 1)
(1, 1, 3)
---------
 2  2  3
```

即三个整数数组广播之后的 shape 属性为(2,2,3)，这也就是下标运算所得到的结果数组的维数：

```
>>> b.shape
(2, 2, 3)
```

可以使用 broadcast_arrays()查看广播之后的数组：

```
>>> ind0, ind1, ind2 = np.broadcast_arrays(i0, i1, i2)
>>> ind0
array([[[1, 2, 1],
        [0, 1, 0]],
       [[1, 2, 1],
        [0, 1, 0]]])
>>> ind1
array([[[0, 0, 0],
        [0, 0, 0]],
       [[1, 1, 1],
        [1, 1, 1]]])
>>> ind2
array([[[2, 3, 2],
        [2, 3, 2]],
       [[2, 3, 2],
        [2, 3, 2]]])
```

对于数组 b 中的任意一个元素 b[i,j,k]，它是数组 a 中经过 ind0、ind1 和 ind2 进行下标转换之后的值：

```
>>> i,j,k = 0,1,2
>>> b[i,j,k]
2
>>> a[ind0[i,j,k],ind1[i,j,k],ind2[i,j,k]]
2
>>> i,j,k = 1,1,1
>>> b[i,j,k]
28
>>> a[ind0[i,j,k],ind1[i,j,k],ind2[i,j,k]]
28
```

下面考虑 N_s 不为 0 的情况。当存在切片下标时，情况就变得更加复杂了。可以细分为两种情况：下标元组中的整数数组之间没有切片，即整数数组只有一个或者是连续的。这时结果数组的 shape 属性为：将原始数组的 shape 属性中整数数组所占据的部分替换为它们广播之后的 shape 属性。例如，假设原始数组 a 的 shape 属性为(3,4,5)，i0 和 i1 广播之后的形状为(2,2,3)，则 a[1:3,i0,i1]的形状为(2,2,2,3)：

```
>>> c=a[1:3, i0, i1]
>>> c.shape
(2, 2, 2, 3)
```

其中，数组 c 的 shape 属性中的第一个 2 是切片“1:3”的长度，后面的(2,2,3)则是 i0 和 i1 广播之后数组的形状：

```
>>> ind0, ind1 = np.broadcast_arrays(i0, i1)
>>> ind0.shape
    (2, 2, 3)
>>> i,j,k = 1,1,2
>>> c[:,i,j,k]
    array([21, 41])
>>> a[1:3,ind0[i,j,k],ind1[i,j,k]] # 和c[:,i,j,k]的值相同
    array([21, 41])
```

如果下标元组中的整数数组不是连续的，那么结果数组的 shape 属性为整数数组广播之后的形状后面再加上切片元素对应的形状。例如，a[i0,:,i1]的 shape 属性为(2,2,3,4)。其中，(2,2,3)是 i0 和 i1 广播之后的形状，而 4 是数组 a 第 1 轴的长度：

```
>>> d = a[i0, :, i1]
>>> d.shape
    (2, 2, 3, 4)
>>> i,j,k = 1,1,2
>>> d[i,j,k,:]
    array([ 1,  6, 11, 16])
>>> a[ind0[i,j,k],:,ind1[i,j,k]]
    array([ 1,  6, 11, 16])
```

2.3.3　一个复杂的例子

下面让我们用所学的下标存取的知识，解决在 NumPy 邮件列表中提出的一个比较经典的问题，此问题的原文链接地址为：



http://mail.scipy.org/pipermail/numpy-discussion/2008-July/035764.html
NumPy 邮件列表中的原文链接

我们对问题进行一些简化，提问者想要实现的下标运算是：有一个形状为(I, J, K)的三维数组 v 和一个形状为(I, J)的二维数组 idx，idx 的每个值都是 0 到 K-L 的整数。他想通过下标运算得到一个数组 r，对于第 0 轴和第 1 轴的每个下标 i 和 j 都满足下面条件：

```
r[i,j,:] = v[i,j,idx[i,j]:idx[i,j]+L]
```

如图 2-6 所示[⑤]，左图中不透明的方块是我们希望获取的部分，通过下标运算之后将得到右图所示的数组。

⑤ 绘制此图的源程序为“numpy_array_index_demo.py”。

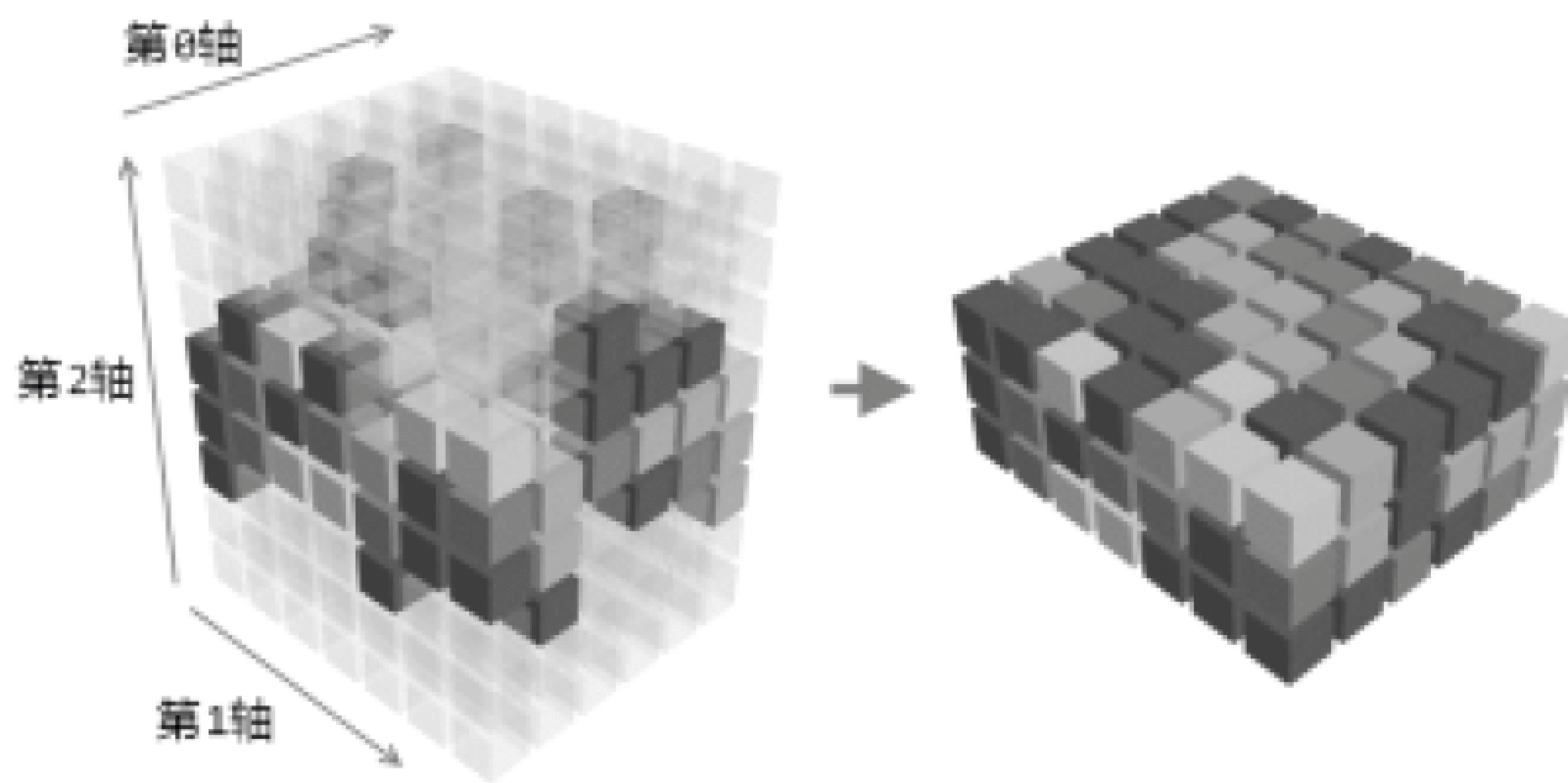


图 2-6　三维数组下标运算问题的示意图

首先创建一个方便调试的数组 v，它在第 2 轴上每一层的值就是该层的高度，即 v[:,:,i]的所有元素的值都为 i。然后随机产生数组 idx，它的每个元素的取值都在 0 到 K-L 之间：



numpy_array_index.py
三维数组下标运算问题

```
>>> I, J, K, L = 6, 7, 8, 3
>>> _, _, v = np.mgrid[:I, :J, :K]
>>> idx = np.random.randint(0, K-L, size=(I,J))
```

然后用数组 idx 创建第 2 轴的下标数组 idx_k，它是一个形状为(I,J,L)的三维数组。它的第 2 轴上的每一层的值都等于 idx 数组加上层的高度，即"idx_k[:,:,i] = idx[:,:]+i"：

```
>>> idx_k = idx.reshape(I,J,1) + np.arange(L)
>>> idx_k.shape
(6, 7, 3)
```

然后分别创建第 0 轴和第 1 轴的下标数组，它们的 shape 属性分别为(I,1,1)和(1,J,1)：

```
>>> idx_i, idx_j, _ = np.ogrid[:I, :J, :K]
```

使用 idx_i、idx_j、idx_k 对数组 v 进行下标运算即可得到结果：

```
>>> r = v[idx_i, idx_j, idx_k]
>>> i, j = 2, 3  # 验证结果，读者可以修改为使用循环测试
>>> r[i,j,:]
array([4, 5, 6])
>>> v[i,j,idx[i,j]:idx[i,j]+L]
array([4, 5, 6])
```

2.3.4 布尔数组作为下标

当使用布尔数组直接作为下标对象或者元组下标对象中有布尔数组时，都相当于用nonzero()将布尔数组转换成一组整数数组，然后使用整数数组进行下标运算。

nonzeros(a)返回数组 a 中值不为零的元素的下标，它的返回值是一个长度为 a.ndim(数组 a 的轴数)的元组，元组的每个元素都是一个整数数组，其值为非零元素的下标在对应轴上的值。例如，对于一维布尔数组 b1，nonzero(b1)得到的是一个长度为 1 的元组，它表示 b1[0]和 b1[2]的值不为 0(False)。

```
>>> b1 = np.array([True, False, True, False])
>>> np.nonzero(b1)
    (array([0, 2]),)
```

对于二维数组 b2，nonzero(b2)得到的是一个长度为 2 的元组。它的第 0 个元素是数组 a 中值不为 0 的元素的第 0 轴的下标，第 1 个元素则是第 1 轴的下标，因此从下面的结果可知 b2[0,0]、b[0,2]和 b2[1,0]的值不为 0：

```
>>> b2 = np.array([[True, False, True], [True, False, False]])
>>> np.nonzero(b2)
    (array([0, 0, 1]), array([0, 2, 0]))
```

当布尔数组直接作为下标时，相当于使用由 nonzero()转换之后的元组作为下标对象：

```
>>> a = np.arange(3*4*5).reshape(3,4,5)
>>> a[b2]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
>>> a[np.nonzero(b2)]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
```

当下标对象是元组，并且其中有布尔数组时，相当于将布尔数组展开为由 nonzeros()转换之后的各个整数数组：

```
>>> a[1:3, b2]
array([[20, 22, 25],
       [40, 42, 45]])
>>> a[1:3, np.nonzero(b2)[0], np.nonzero(b2)[1]]
array([[20, 22, 25],
       [40, 42, 45]])
```

2.4 庞大的函数库

除了前面介绍的 ndarray 数组对象和 ufunc 函数之外，NumPy 还提供了大量对数组进行处理的函数。充分利用这些函数，能够简化程序的逻辑，提高运算速度。本节通过一些较为常用的例子，说明它们的一些使用技巧和注意事项。

2.4.1 求和、平均值、方差

sum()计算数组元素之和，也可以对列表、元组等和数组类似的序列进行求和。当数组是多维时，它计算数组中所有元素的和：

```
>>> a = np.random.randint(0,10,size=(4,5))
>>> a
array([[7, 1, 9, 6, 3],
       [5, 1, 3, 8, 2],
       [9, 8, 9, 4, 0],
       [9, 5, 1, 7, 0]])
>>> np.sum(a)
97
```

如果指定 axis 参数，求和运算将沿着指定的轴进行。在上面的例子中，数组 a 的第 0 轴长度为 4，第 1 轴长度为 5。如果 axis 参数为 1，就对每行上的 5 个数求和，所得的结果是长度为 4 的一维数组。如果参数 axis 为 0，就对每列上的 4 个数求和，结果是长度为 5 的一维数组。即，结果数组的形状是原始数组的形状除去其第 axis 个元素：

```
>>> np.sum(a, axis=1)
array([26, 19, 30, 22])
>>> np.sum(a, axis=0)
array([30, 15, 22, 25,  5])
```

上面的例子将产生一个新的数组来保存求和结果，如果希望将结果直接保存到另外一个数组中，可以和 ufunc 函数一样使用 out 参数指定输出数组，它的形状必须和结果数组的形状相同。

sum()默认使用和数组的元素类型相同的累加变量进行计算，如果元素类型为整数，就使用系统的默认整数类型作为累加变量。在 32 位系统中，累加变量的类型为 32 bit 整型。因此对整数数组进行累加时可能会出现溢出问题，即数组元素的总和超过了累加变量的取值范围。而对很大的单精度浮点数类型数组进行计算时，也可能出现精度不够的现象，这时可以通过 dtype 参数指定累加变量的类型。在下面的例子中，我们对一个元素都为 1.1 的单精度数组进行求和，比较单精度累加变量和双精度累加变量的计算结果：

```
>>> b = np.ones(1000000, dtype=np.float32) * 1.1 # 创建一个很大的单精度浮点数数组
>>> b # 1.1 无法使用浮点数精确表示，存在一些误差
array([ 1.10000002,  1.10000002,  ..., 1.10000002], dtype=float32)
>>> np.sum(b) # 使用单精度累加变量进行累加计算，误差将越来越大
1110920.5
>>> np.sum(b, dtype=np.double) # 使用双精度浮点数则能够得到正确的值
1100000.0238418579
```

mean()用于求数组的平均值，也可以通过 axis 参数指定求平均值的轴，通过 out 参数指定输出数组。和 sum()不同的是，对于整数数组，它使用双精度浮点数进行计算，而对于其他类型的数组，则使用和数组元素类型相同的累加变量进行计算：

```
>>> np.mean(a,axis=1) # 整数数组使用双精度浮点数进行计算
array([ 5.2,  3.8,  6. ,  4.4])
>>> np.mean(b) # 单精度浮点数使用单精度浮点数进行计算
1.1109205
>>> np.mean(b, dtype=np.double) #用双精度浮点数计算平均值
1.1000000238418579
```

除此之外，average()也可以对数组进行平均计算。它没有 out 和 dtype 参数，但有一个指定每个元素权值的 weights 参数，请读者在 IPython 中输入“np.average?”来查看使用说明。

std()和 var()分别计算数组的标准差和方差，有 axis、out 及 dtype 等参数。

2.4.2　最值和排序

用 min()和 max()可以计算数组的最大值和最小值，而 ptp()计算最大值和最小值之间的差。它们都有 axis 和 out 两个参数。这些参数的用法和 sum()相同，这里就不再多举例了，请读者自行查看函数的文档。

用 argmax()和 argmin()可以求最大值和最小值的下标。如果不指定 axis 参数，就返回平坦化之后的数组下标，例如：

```
>>> np.argmax(a) #找到数组 a 中最大值的下标，有多个最值时得到第一个最值的下标
2
>>> a.ravel()[2] # 求平坦化之后的数组中的第二个元素
9
```

可以通过 unravel_index()将一维下标转换为多维数组中的下标，它的第一个参数为一维下标值，第二个参数是多维数组的形状：

```
>>> idx = np.unravel_index(2, a.shape)
>>> idx
(0, 2)
>>> a[idx]
9
```

当使用 axis 参数时，可以沿着指定的轴计算最大值的下标。例如下面的结果表示，在数组 a 中，第 0 行中最大值的下标为 2，第 1 行中最大值的下标为 3：

```
>>> idx = np.argmax(a, axis=1)
>>> idx
array([2, 3, 0, 0])
```

下面的语句使用 idx 选择出每行的最大值：

```
>>> a[xrange(a.shape[0]), idx]
array([9, 8, 9, 9])
```

数组的 sort()方法用于对数组进行排序，它将改变数组的内容。而 sort()函数则返回一个新数组，不改变原始数组。它们的 axis 参数默认值都为-1，即沿着数组的最后一个轴进行排序。sort()函数的 axis 参数可以设置为 None，此时它将得到平坦化之后进行排序的新数组。

```
>>> np.sort(a) # 对每行的数据进行排序
array([[1, 3, 6, 7, 9],
       [1, 2, 3, 5, 8],
       [0, 4, 8, 9, 9],
       [0, 1, 5, 7, 9]])
>>> np.sort(a, axis=0) # 对每列的数据进行排序
array([[5, 1, 1, 4, 0],
       [7, 1, 3, 6, 0],
       [9, 5, 9, 7, 2],
       [9, 8, 9, 8, 3]])
```

argsort()返回数组的排序下标，axis 参数的默认值为-1：

```
>>> idx = np.argsort(a)
>>> idx
array([[1, 4, 3, 0, 2],
       [1, 4, 2, 0, 3],
       [4, 3, 1, 0, 2],
       [4, 2, 1, 3, 0]])
```

为了使用 idx 计算排序之后的数组，即 np.sort(a)的结果，需要产生第 0 轴的下标。下面使用 ogrid 对象产生第 0 轴的下标 x：

```
>>> x, _ = np.ogrid[:a.shape[0],:a.shape[1]]
>>> x
array([[0],
       [1],
       [2],
```

```
        [3]])
>>> a[x, idx]     #和 np.sort(a)的结果相同
array([[1, 3, 6, 7, 9],
       [1, 2, 3, 5, 8],
       [0, 4, 8, 9, 9],
       [0, 1, 5, 7, 9]])
```

使用这种方法可以对两个相互关联的数组进行排序，即从数组 a 产生排序下标数组，然后使用它对数组 b 进行排序。排序相关的函数或方法还可以通过 kind 参数指定排序算法，对于结构数组可以通过 order 参数指定排序所使用的字段。

用 median()可以获得数组的中值，即对数组进行排序之后，位于数组中间位置的值。当长度是偶数时，得到正中间两个数的平均值。它也可以指定 axis 和 out 参数：

```
>>> np.median(a, axis=0)
array([ 8. ,  3. ,  6. ,  6.5,  1. ])
```

2.4.3 多项式函数

多项式函数是变量的整数次幂与系数的乘积之和，可以用下面的数学公式表示：

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots\ldots + a_2 x^2 + a_1 x + a_0$$

由于多项式函数只包含加法和乘法运算，因此它很容易计算，并且可以用于计算其他数学函数的近似值。多项式函数的应用非常广泛，例如在嵌入式系统中经常会用它计算正弦、余弦等函数。在 NumPy 中，多项式函数的系数可以用一维数组表示，例如 $f(x)=x^3-2x+1$ 可以用下面的数组表示，其中 a[0]是最高次的系数，a[-1]是常数项，注意 x^2 的系数为 0。

```
>>> a = np.array([1.0, 0, -2, 1])
```

可以用 poly1d()将系数转换为 poly1d(一元多项式)对象，此对象可以像函数一样调用，它返回多项式函数的值：

```
>>> p = np.poly1d(a)
>>> type(p)
<class 'numpy.lib.polynomial.poly1d'>
>>> p(np.linspace(0, 1, 5))
array([ 1.      ,  0.515625,  0.125   , -0.078125,  0.      ])
```

对 poly1d 对象进行加减乘除运算相当于对相应的多项式函数进行计算。例如：

```
>>> p + [-2, 1] # 和 p + np.poly1d([-2, 1]) 相同
poly1d([ 1.,  0., -4.,  2.])
>>> p * p # 两个 3 次多项式相乘得到一个 6 次多项式
poly1d([ 1.,  0., -4.,  2.,  4., -4.,  1.])
```

```
>>> p / [1, 1] # 除法返回两个多项式，分别为商式和余式
(poly1d([ 1., -1., -1.]), poly1d([ 2.]))
```

由于多项式的除法不一定能正好整除，因此它返回除法所得到的商式和余式。上面的例子中，商式为 x^2-x-1，余式为 2。因此将商式和被除式相乘后，再加上余式就等于原来的 p：

```
>>> p == np.poly1d([ 1., -1., -1.]) * [1,1] + 2
>>> True
```

多项式对象的 deriv()和 integ()方法分别计算多项式函数的微分和积分：

```
>>> p.deriv()
   poly1d([ 3.,  0., -2.])
>>> p.integ()
   poly1d([ 0.25,  0.  , -1.  ,  1.  ,  0.  ])
>>> p.integ().deriv() == p
True
```

多项式函数的根可以使用 roots()函数计算：

```
>>> r = np.roots(p)
>>> r
array([-1.61803399,  1.        ,  0.61803399])
>>> p(r) # 将根带入多项式计算，得到的值近似为 0
array([ -4.21884749e-15,  -4.44089210e-16,  -2.22044605e-16])
```

而 poly()函数可以将根转换回多项式的系数：

```
>>> np.poly(r)
array([1.00000000e+00, 9.99200722e-16, -2.00000000e+00, 1.00000000e+00])
```

除了使用多项式对象之外，还可以直接使用 NumPy 提供的多项式函数对表示多项式系数的数组进行运算。可以在 IPython 中使用自动补全查看函数名：

```
>>> np.poly # 按 Tab 键
np.poly    np.polyadd np.polydiv np.polyint np.polysub
np.poly1d  np.polyder np.polyfit np.polymul np.polyval
>>> np.polymul([1,1],[1,1])
array([1, 2, 1])
```

其中：polyfit()函数可以对一组数据使用多项式函数进行拟合，找到和这组数据最接近的多项式的系数。下面的程序用它计算 $-\pi/2 \sim \pi/2$ 区间与 $sin(x)$函数最接近的多项式的系数：

numpy_polyfit.py
用多项式函数拟合正弦函数

```
x = np.linspace(-np.pi/2, np.pi/2, 1000) ❶
y = np.sin(x) ❷

for deg in [3,5,7]:
    a = np.polyfit(x, y, deg) ❸
    error = np.abs(np.polyval(a, x)-y) ❹
    print "poly %d:" % deg, a
    print "max error of order %d:" % deg , np.max(error)
```

❶首先通过 linspace()将 $-\pi/2 \sim \pi/2$ 区间等分为 1000 -1 等份，❷并计算拟合目标函数 $sin(x)$ 的值。❸将表示目标函数的数组传递给 polyfit()进行拟合，第三个参数 deg 为多项式函数的最高阶数。polyfit()所得到的多项式和目标函数在给定的 1000 个点之间的误差最小，polyfit()返回多项式的系数数组。❹使用 polyval()计算多项式函数的值，并计算与目标函数的差的绝对值。

由于正弦函数是奇函数，因此只使用奇数阶的多项式对其进行拟合。为了直观地观察误差，我们使用图 2-7 显示拟合之后的各阶多项式与正弦函数之间的误差，请注意图中 Y 轴为对数坐标(见封二彩插)。下面是程序的输出，可以看到由于正弦函数是一个奇函数，因此拟合的多项式系数中偶次项的系数接近于 0。

图 2-7　各阶多项式函数拟合正弦函数的误差

```
poly 3: [ -1.45021094e-01  -6.39518172e-16   9.88749145e-01  -4.29811537e-15]
max error of order 3: 0.00894699376708
poly 5: [  7.57279944e-03  -2.50656614e-17  -1.65823793e-01  -2.72346001e-18
   9.99770071e-01   7.17317591e-18]
max error of order 5: 0.00015740861417
poly 7: [ -1.84445514e-04   3.70441786e-17   8.30942467e-03  -1.24633291e-16
  -1.66651593e-01   7.40085118e-17   9.99997468e-01  -8.11201916e-18]
max error of order 7: 1.52682558119e-06
```

2.4.4 分段函数

在 2.2.3 节中我们介绍了如何使用 frompyfunc()函数计算三角波形。由于三角波形是分段函数，需要根据自变量的取值范围决定计算函数值的公式，因此无法直接通过 ufunc 函数计算它。NumPy 提供了一些计算分段函数的方法。

numpy_piecewise.py
用 where()和 piecewise()计算三角波形

Python 2.6 中新增了如下的判断表达式语法，当 condition 条件为 True 时，表达式的值为 y，否则为 z：

```
x = y if condition else z
```

在 NumPy 中，where()函数可以看做判断表达式的数组版本：

```
x = where(condition, y, z)
```

其中，condition、y 和 z 都是数组，它的返回值是一个形状与 condition 相同的数组。当 condition 中的某个元素为 True 时，数组 x 中对应下标的值从数组 y 获取，否则从数组 z 获取：

```
>>> x = np.arange(10)
>>> np.where(x<5, 9-x, x)
array([9, 8, 7, 6, 5, 5, 6, 7, 8, 9])
```

如果 y 和 z 是单个的数值或者它们的形状与 condition 的不同，将先通过广播运算使其形状一致：

```
>>> np.where(x>6, 2*x, 0)
array([ 0,  0,  0,  0,  0,  0,  0, 14, 16, 18])
```

使用 where()很容易计算 2.2.3 节介绍的三角波形。

```
def triangle_wave(x, c, c0, hc):
    x = x - x.astype(np.int) # 三角波的周期为 1，因此只取 x 坐标的小数部分进行计算
    return np.where(x>=c, 0, np.where(x<c0, x/c0*hc, (c-x)/(c-c0)*hc))
```

由于三角波形分为三段，因此需要两个嵌套的 where()进行计算。由于所有的运算和循环都在 C 语言级别完成，因此它的计算效率比 frompyfunc()高。

随着分段函数的分段数量的增加，需要嵌套更多层 where()，但这样做不便于程序的编写和阅读。可以用 select()解决这个问题，它的调用形式如下：

```
select(condlist, choicelist, default=0)
```

其中，condlist 是一个长度为 N 的布尔数组列表，choicelist 是一个长度为 N 的储存候选值的数组列表，所有数组的长度都为 M。如果列表元素不是数组而是单个数值，那么它相当于元素值都相同且长度为 M 的数组。

对于从 0 到 M-1 的数组下标 i，从布尔数组列表中找出满足条件“condlist[j][i]==True”的 j 的最小值，则“out[i]=choicelist[j][i]”，其中 out 是 select()的返回数组。可以使用 select()计算三角波形：

```
def triangle_wave2(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.select([x>=c, x<c0, True], [0, x/c0*hc, (c-x)/(c-c0)*hc])
```

由于分段函数分为三段，因此每个列表都有三个元素。choicelist 的最后一个元素为 True，表示前面所有条件都不满足时，将使用 choicelist 的最后一个数组中的值。也可以用 default 参数指定条件都不满足时的候选值数组：

```
return np.select([x>=c, x<c0], [0, x/c0*hc], default=(c-x)/(c-c0)*hc)
```

但是 where()和 select()的所有参数都需要在调用它们之前完成计算，因此 NumPy 会计算下面 4 个数组：

```
x>=c, x<c0, x/c0*hc, (c-x)/(c-c0)*hc
```

在计算时还会产生许多保存中间结果的数组，因此如果输入的数组 x 很大，将会发生大量的内存分配和释放。

为了解决这个问题，NumPy 提供了 piecewise()专门用于计算分段函数，它的调用参数如下：

```
piecewise(x, condlist, funclist)
```

参数 x 是一个保存自变量值的数组。condlist 是一个长度为 M 的布尔数组列表，其中的每个布尔数组的长度都和数组 x 相同。funclist 是一个长度为 M 或 M+1 的函数列表，这些函数的输入和输出都是数组。它们计算分段函数中的每个片段。如果不是函数而是数值，就相当于返回此数值的函数。每个函数与 condlist 中下标相同的布尔数组对应，如果 funclist 的长度为 M+1，那么最后一个函数对应于所有条件都为 False 时。下面是使用 piecewise()计算三角波形的程序：

```
def triangle_wave3(x, c, c0, hc):
    x = x - x.astype(np.int)
    return np.piecewise(x,
        [x>=c, x<c0],
        [0,  # x>=c
        lambda x: x/c0*hc, # x<c0
        lambda x: (c-x)/(c-c0)*hc])  # else
```

使用 piecewise()的好处在于它只计算需要计算的值。因此在上面的例子中，表达式“x/c0*hc”和“(c-x)/(c-c0)*hc”只对输入数组 x 中满足条件的部分进行计算。请读者思考一下如何验证上例中每个函数所计算的数组的长度，解答在“numpy_piecewise.py”中。

2.4.5 统计函数

unique()返回其参数数组中所有不同的值，并且按照从小到大的顺序排列。它有两个可选参数：

- return_index：Ture 表示同时返回原始数组中的下标。
- return_inverse：True 表示返回重建原始数组用的下标数组。

下面通过实例介绍 unique()的用法。首先用 randint()创建含有 10 个元素、值在 0 到 9 范围之内的随机整数数组：

```
>>> a = np.random.randint(0,5,10)
>>> a
array([1, 1, 9, 5, 2, 6, 7, 6, 2, 9])
```

通过 unique(a)可以找到数组 a 中所有的整数，并按照顺序排列：

```
>>> np.unique(a)
array([1, 2, 5, 6, 7, 9])
```

如果参数 return_index 为 True，就返回两个数组，第二个数组是第一个数组在原始数组中的下标：

```
>>> x, idx = np.unique(a, return_index=True)
>>> x
array([1, 2, 5, 6, 7, 9])
>>> idx
array([0, 4, 3, 5, 6, 2])
```

数组 idx 保存的是数组 x 中每个元素在数组 a 中的下标：

```
>>> a[idx]
array([1, 2, 5, 6, 7, 9])
```

如果参数 return_inverse 为 True，那么返回的第二个数组是原始数组 a 中每个元素在数组 x 中的下标：

```
>>> x, ridx = np.unique(a, return_inverse=True)
>>> ridx
array([0, 0, 5, 2, 1, 3, 4, 3, 1, 5])
```

```
>>> all(x[ridx] == a) # 原始数组 a 和 x[ridx]完全相同
True
```

bincount()对整数数组中各个元素出现的次数进行统计，它要求数组中所有元素都是非负的。其返回数组中第 i 个元素的值表示整数 i 在参数数组中出现的次数。

```
>>> np.bincount(a)
array([0, 2, 2, 0, 0, 1, 2, 1, 0, 2])
```

由上面的结果可知，在数组 a 中有两个 1、两个 2、一个 5、两个 6、一个 7 和两个 9，而 0、3、4、8 等数没有在数组 a 中出现。

通过 weights 参数可以指定每个数对应的权值。当指定 weights 参数时，bincount(x, weights=w)返回数组 x 中每个整数所对应的 w 中的权值之和。用文字解释比较难以理解，下面我们看一个实例：

```
>>> x = np.array([0  ,   1,   2,   2,   1,   1,   0])
>>> w = np.array([0.1, 0.3, 0.2, 0.4, 0.5, 0.8, 1.2])
>>> np.bincount(x, w)
array([ 1.3,  1.6,  0.6])
```

上面的结果中，1.3 是数组 x 中 0 所对应的 w 中的值(0.1 和 1.2)的和，1.6 是 1 所对应的 w 中的值(0.3、0.5 和 0.8)的和，而 0.6 是 2 所对应的 w 中的值(0.2 和 0.4)的和。如果要求平均值，可以用求和结果与次数相除：

```
>>> np.bincount(x,w)/np.bincount(x)
array([ 0.65      ,  0.53333333,  0.3       ])
```

histogram()对一维数组进行直方图统计，其参数列表如下：

```
histogram(a, bins=10, range=None, normed=False, weights=None)
```

其中，a 是保存待统计数据的数组，bins 指定统计的区间个数，即对统计范围的等分数。range 是一个长度为 2 的元组，表示统计范围的最小值和最大值，默认值为 None，表示范围由数据的范围决定，即(a.min(), a.max())。当 normed 参数为 False 时，函数返回数组 a 中的数据在每个区间的个数，否则对个数进行正规化处理，使它等于每个区间的概率密度。weights 参数和 bincount()的类似。

histogram()返回两个一维数组——hist 和 bin_edges，第一个数组是每个区间的统计结果，第二个数组长度为 len(hist)+1，每两个相邻的数值构成一个统计区间。下面我们看一个例子：

```
>>> a = np.random.rand(100)
>>> np.histogram(a,bins=5,range=(0,1))
(array([20, 26, 20, 16, 18]), array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1.])
```

首先创建了一个包含 100 个元素的一维随机数组 a，取值范围在 0 到 1 之间。然后用 histogram()对数组 a 中的数据进行直方图统计。结果显示有 20 个元素的值在 0 到 0.2 之间，26 个元素的值在 0.2 到 0.4 之间。读者可以尝试用 rand()创建更大的随机数组，由统计结果可知每个区间出现的次数近似相等，因此 rand()所创建的随机数在 0 到 1 范围之间是平均分布的。

如果需要统计的区间长度不等，可以将表示区间分隔位置的数组传递给 bins 参数，例如：

```
>>> np.histogram(a, bins=[0, 0.4, 0.8, 1.0], range=(0,1))
(array([46, 36, 18]), array([ 0. ,  0.4,  0.8,  1. ]))
```

结果表示：0 到 0.4 之间有 46 个值，0.4 到 0.8 之间有 36 个值。

如果用 weights 参数指定了数组 a 中每个元素对应的权值，那么 histogram()将对区间中数值对应的权值进行求和。下面看一个使用 histogram()统计男青少年年龄和身高的例子。

height.csv
100 名年龄在 7 到 20 岁之间的男性青少年的身高统计数据

首先用 loadtxt()从数据文件载入数据，关于文件存取方面的内容后面还会详细介绍：

```
>>> d = np.loadtxt("height.csv", delimiter=",")
>>> d.shape
(100, 2)
```

在数组 d 中，第 0 列是年龄，第 1 列是身高。可以看到年龄的范围在 7 到 20 之间：

```
>>> np.min(d[:,0])
7.0999999999999996
>>> np.max(d[:,0])
19.899999999999999
```

下面对数据进行统计，sums 是每个年龄段的身高总和，cnts 是每个年龄段的数据个数，因此很容易计算出每个年龄段的平均身高：

```
>>> sums = np.histogram(d[:,0],bins=range(7,21),range=(7,20), weights=d[:,1])[0]
>>> cnts = np.histogram(d[:,0],bins=range(7,21),range=(7,20))[0]
>>> sums/cnts
array([ 125.96      ,  132.06666667,  137.82857143,  143.8       ,
        148.14      ,  153.44      ,  162.15555556,  166.86666667,
        172.83636364,  173.3       ,  175.275     ,  174.19166667,  175.075     ])
```

histogram2d()和 histogramdd()用来对二维和 N 维数据进行直方图统计，我们将在第 12 章对 histogram2d()进行详细介绍。

2.5 线性代数

NumPy 和 MATLAB 不同，对于多维数组的运算，默认情况下并不使用矩阵运算。如果读者希望对数组进行矩阵运算，可以调用相应的函数。

matrix 对象

NumPy 库提供了 matrix 类，使用 matrix 类创建的是矩阵对象，它们的加减乘除运算默认采用矩阵方式计算，因此用法和 MATLAB 十分类似。但是由于 NumPy 中同时存在 ndarray 和 matrix 对象，因此用户很容易将两者弄混。这有违 Python 的“显式优于隐式”的原则，因此并不推荐在较复杂的程序中使用 matrix 对象。下面是使用 matrix 的一个例子：

```
>>> a = np.matrix([[1,2,3],[5,5,6],[7,9,9]])
>>> a*a**-1
matrix([[ 1.00000000e+00,   1.66533454e-16,  -8.32667268e-17],
        [ -2.77555756e-16,   1.00000000e+00,  -2.77555756e-17],
        [  1.66533454e-16,   5.55111512e-17,   1.00000000e+00]])
```

因为 a 是用 matrix()创建的矩阵对象，因此乘法和幂运算符都变成了矩阵运算，于是上面计算的是矩阵 a 与其逆矩阵的乘积，结果是一个单位矩阵。

2.5.1 各种乘积运算

矩阵的乘积可以使用 dot()来计算。对于二维数组，它计算的是矩阵乘积；对于一维数组，它计算的是内积。当需要将一维数组当作列矢量或行矢量进行矩阵运算时，推荐先使用 reshape()将一维数组转换为二维数组：

```
>>> a = array([1, 2, 3])
>>> a.reshape((-1,1))
array([[1],
       [2],
       [3]])
>>> a.reshape((1,-1))
array([[1, 2, 3]])
```

除了使用 dot()计算乘积之外，NumPy 还提供了 inner()和 outer()等多种计算乘积的函数。这些函数计算乘积的方式不同，尤其是当参数是多维数组时，很容易搞混淆。

- dot：对于两个一维数组，计算的是这两个数组对应下标元素的乘积和，数学上称之为内积。对于二维数组，计算的是两个数组的矩阵乘积；对于多维数组，它的通用

计算公式如下。即结果数组中的每个元素都是：数组 a 最后一维上的所有元素与数组 b 倒数第二维上的所有元素的乘积和。

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

下面以两个 3 维数组的乘积演示 dot 乘积的计算结果。首先创建两个 3 维数组，这两个数组的最后两维满足矩阵乘积的条件：

```
>>> a = np.arange(12).reshape(2,3,2)
>>> b = np.arange(12,24).reshape(2,2,3)
>>> c = np.dot(a,b)
```

dot 乘积的结果 c 是数组 a 和 b 的多个子矩阵的乘积：

```
>>> np.alltrue( c[0,:,0,:] == np.dot(a[0],b[0]) )
True
>>> np.alltrue( c[1,:,0,:] == np.dot(a[1],b[0]) )
True
>>> np.alltrue( c[0,:,1,:] == np.dot(a[0],b[1]) )
True
>>> np.alltrue( c[1,:,1,:] == np.dot(a[1],b[1]) )
True
```

- inner：和 dot 乘积一样，对于两个一维数组，计算的是这两个数组对应下标元素的乘积和。对于多维数组，它计算的结果数组中的每个元素都是：数组 a 和 b 最后一维的内积，因此数组 a 和 b 最后一维的长度必须相同。

```
inner(a, b)[i,j,k,m] = sum(a[i,j,:]*b[k,m,:])
```

下面是 inner 乘积的演示：

```
>>> a = np.arange(12).reshape(2,3,2)
>>> b = np.arange(12,24).reshape(2,3,2)
>>> c = np.inner(a,b)
>>> c.shape
(2, 3, 2, 3)
>>> c[0,0,0,0] == np.inner(a[0,0],b[0,0])
True
>>> c[0,1,1,0] == np.inner(a[0,1],b[1,0])
True
>>> c[1,2,1,2] == np.inner(a[1,2],b[1,2])
True
```

- outer：只按照一维数组进行计算，如果传入参数是多维数组，先将此数组展平为一维数组，然后再进行运算。outer 乘积计算列向量和行向量的矩阵乘积：

```
>>> np.outer([1,2,3],[4,5,6,7])
array([[ 4,  5,  6,  7],
       [ 8, 10, 12, 14],
       [12, 15, 18, 21]])
```

2.5.2 解线性方程组

对矩阵的一些更高级运算可以在 NumPy 的线性代数模块 linalg 中找到。例如 inv()计算逆矩阵，solve()可以求解多元一次方程组。下面是 solve()的一个例子：

```
>>> a = np.random.rand(10,10)
>>> b = np.random.rand(10)
>>> x = np.linalg.solve(a,b)
>>> np.sum(np.abs(np.dot(a,x) - b))
3.1433189384699745e-15
```

solve()有两个参数 a 和 b。a 是一个 N*N 的二维数组，而 b 是一个长度为 N 的一维数组，solve()找到一个长度为 N 的一维数组 x，使得它们满足 $a \cdot x = \boldsymbol{b}$，即 x 就是这个 N 元一次方程组的解。

lstsq()比 solve()更一般化，它不要求矩阵 a 是正方形的，也就是说方程的个数可以少于、等于或多于未知数的个数。它找到一组解 x，使得 $|b - a \cdot x|$ 最小。我们称得到的结果为最小二乘解，即它使得所有等式的误差的平方和最小。下面以求解离散卷积的逆运算为例，介绍 lstsq()的用法。

首先简单介绍一下离散卷积的相关知识和计算方法。对于一个离散的线性时不变系统 h，如果它的输入是 x，那么其输出 y 可以用 x 和 h 的卷积表示：

$$y = x * h$$

离散卷积的计算公式如下：

$$y[n] = \sum h[m]x[n-m]$$

假设 h 的长度为 N，x 的长度为 M，则卷积计算得到的 y 的长度将为 N+M-1。它的每个值都是按照下面的公式计算得到的：

```
y[0] = h[0]*x[0]
y[1] = h[0]*x[1] + h[1]*x[0]
y[2] = h[0]*x[2] + h[1]*x[1] + h[2]*x[0]
y[3] = h[0]*x[3] + h[1]*x[2] + h[2]*x[1]
...
```

```
y[N+M-1] = h[N-1]*x[M-1]
```

所谓卷积的逆运算，是指：假设已知 x 和 y，需要求解 h。由于 h 的长度为 N，于是有 N 个未知数，而由于 y 的长度为 N+M－1，因此这 N 个未知数需要满足 N+M－1 个线性方程。由于方程数比未知数多，卷积的逆运算不一定有精确解，因此问题就变成了找到一组 h，使得它和 x 的卷积与 y 之间的误差最小，显然它就是最小二乘解。下面的程序演示了如何使用 lstsq() 计算卷积的逆运算。

numpy_lstsq.py
用最小二乘法求卷积的逆运算

```
import numpy as np
from numpy.lib.stride_tricks import as_strided

def make_data(M, N, noise_scale): ❶
    x = np.random.standard_normal(M)
    h = np.random.standard_normal(N)
    y = np.convolve(x, h)
    yn = y + np.random.standard_normal(len(y)) * noise_scale * np.max(y)
    return x, yn, h

def solve_h(x, y, N): ❷
    X = as_strided(x, shape=(len(x)-N+1, N), strides=(x.itemsize, x.itemsize)) ❸
    Y = y[N-1:len(x)] ❹
    h = np.linalg.lstsq(X, Y) ❺
    return h[0][::-1] ❻
```

❶首先 make_data()创建需要的数据，它使用 standard_normal()计算正规分布的随机数数组作为 x、y。它返回系统的输入信号 x，以及添加了随机噪声的输出信号 yn。为了和最小二乘法的结果比较，我们同时也输出了系统的系数 h。

❷solve_h()使用最小二乘法计算系统的参数 h，因为通常我们不知道未知系统的系数的长度，因此这里用 N 表示所求系数的长度。

观察前面的卷积方程组可知，在 N+M－1 个方程中，中间的 M－N+1 个方程使用了 h 的所有系数。为了程序计算方便，我们对这 M－N+1 个方程进行最小二乘运算。❸因此根据 h 的长度，需要将一维数组 x 变换成一个形状为(M－N+1, N)的二维数组 X，它的每行相对于上一行都左移了一个元素。这个二维数组可以很容易地使用前面介绍过的 as_strided()得到。❹我们取出输出数组 y 中与数组 X 每行对应的部分，❺然后调用 lstsq()对这 M－N+1 个方程进行最小二乘运算。❻lstsq()返回一个元组，它的第 0 个元素是最小二乘解，注意所得到的结果顺序是颠倒的，因此我们还需要对其进行翻转。

```
x, yn, h = make_data(1000, 100, 0.4)
```

```
H = solve_h(x, yn, 120)
H2 = solve_h(x, yn, 80)
```

接下来对长度为 100 的未知系统系数 h，分别使用长度为 80 和 120 的两个系数对其求最小二乘解。图 2-8 是程序运行得到的结果，虚线是未知系统的系数，实线是最小二乘法的解。由于对系统的输出添加了一些噪声信号，因此二者并不完全吻合。

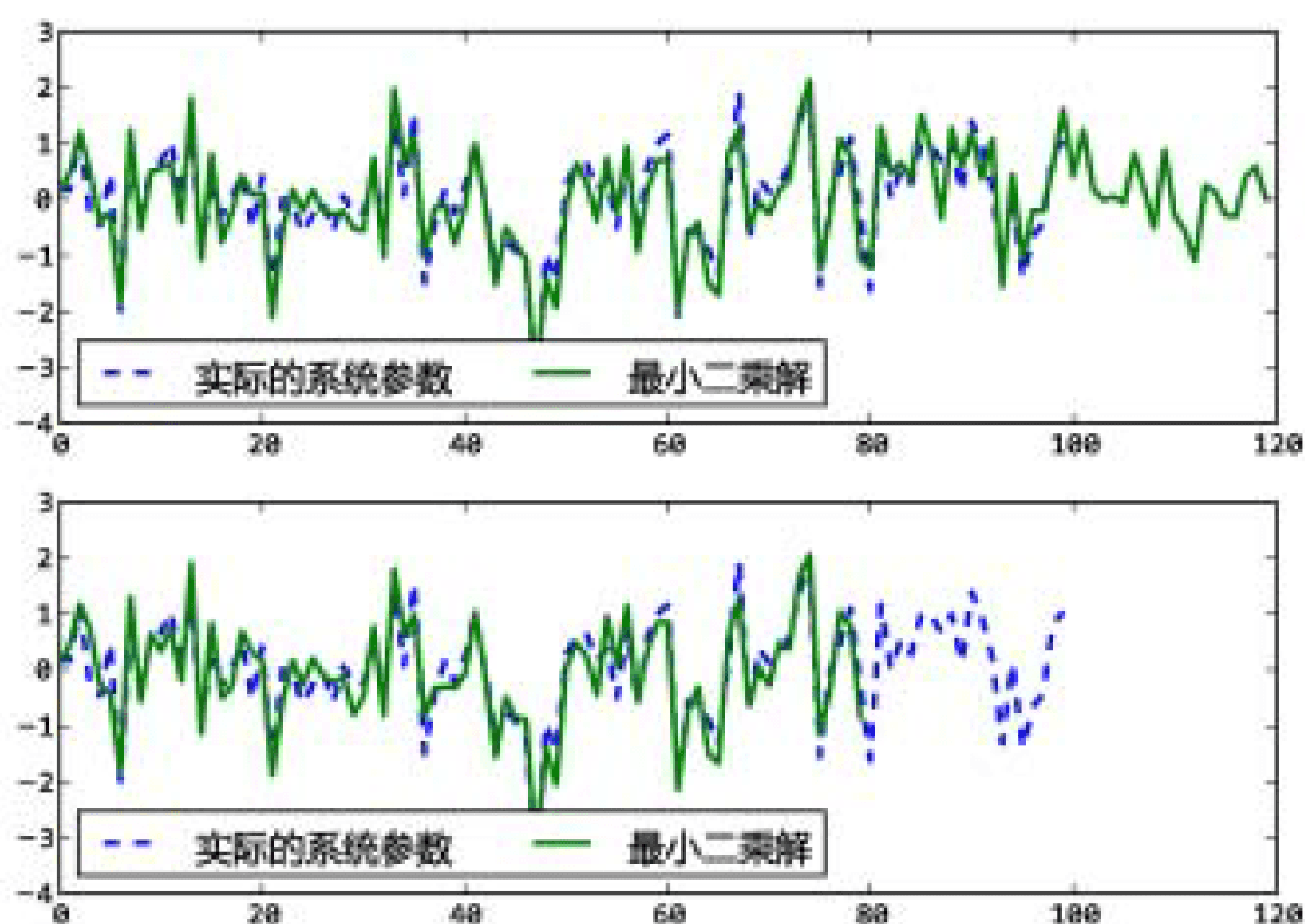


图 2-8　使用最小二乘法求解卷积逆运算：解的长度分别为 120(上)和 80(下)

2.6　掩码数组

在很多情况下，数据可能是不完整的或者存在无效数值。例如对图像数组中某个不规则形状区域中的数据进行处理，或者从传感器中获取的数据存在无效值。在 NumPy 中可以使用掩码数组表示这样的数据，numpy.ma 模块中提供了处理掩码数组的各种函数。这些函数几乎完整地复制了 NumPy 提供的所有函数，为它们增加处理掩码数组的功能。读者可以使用 IPython 的自动完成功能查看它们。

一个掩码数组由一个正常数组和一个布尔数组组成。布尔数组中值为 True 的元素表示正常数组中对应下标的值无效，False 表示有效。下面看一个例子。首先载入 ma 模块，并创建一个随机数组 x 和一个布尔数组 mask。然后调用 ma.array()用这两个数组组成一个掩码数组：

```
>>> import numpy.ma as ma
>>> x = np.array([1,2,3,5,7,4,3,2,8,0])
>>> mask = x < 5 # 小于 5 的值无效
>>> mx = ma.array(x, mask=mask) # 创建掩码数组
>>> mx
masked_array(data = [-- -- -- 5 7 -- -- -- 8 --],
            mask = [ True  True  True False False  True  True  True False  True],
            fill_value = 999999)
```

为了计算掩码数值中有效数组的平均值，可以调用它的 mean()方法，或者调用 ma 模块中的 mean()函数：

```
>>> mx.mean() # 计算掩码数组中有效数值的平均值
6.666666666666667
>>> ma.mean(mx)
6.666666666666667
```

为了验证上面的结果，下面用“~”操作符对 mask 数组取反，并用它作为下标获取数组 x 中的有效数值，然后调用 np.mean()计算平均值：

```
>>> np.mean(x[~mask]) # 使用~mask 作为下标获取有效数值的数组，并求平均值。
6.666666666666667
```

实际上，np.mean()由于直接调用数组对象的 mean()方法，因此用 np.mean()也可以求掩码数组的平均值：

```
>>> np.mean(mx)
6.666666666666667
```

NumPy 中的有些函数不能正确处理掩码数组，为了不引起混淆，建议读者在对掩码数组进行处理时，都使用 ma 模块中定义的函数。例如：

```
>>> np.cov(mx)
array(6.9333333333333336)
>>> ma.cov(mx)
masked_array(data = 2.33333333333,
            mask = False,
      fill_value = 1e+20)
```

通过掩码数组的 data 属性可以获得其数值数组，mask 属性可以获得掩码用的布尔数组。filled()方法返回使用 fill_value 属性表示的填充值替代无效值之后的数组。下面将填充值修改为－1，然后获得填充之后的数组：

```
>>> mx.fill_value = -1
```

```
>>> mx.filled()
array([-1, -1, -1,  5,  7, -1, -1, -1,  8, -1])
```

和一般数组一样，掩码数组也可以使用各种下标对象对其进行存取，例如：

```
>>> mx[0]
masked
>>> mx[:4]
masked_array(
    data = [-- -- -- 5],
    mask = [ True  True  True False],
    fill_value = 999999)
>>> mx[0] = 3 # 填入有效值
>>> mx[0]
3
>>> mx[3] = ma.masked # 用 masked 设置为无效值
```

除了直接指定布尔数组之外，还可以通过 ma 模块提供的各种创建掩码数组的函数，将数组中满足指定条件的元素全部设置为无效。这些函数全部以“masked_”开头，可以通过 IPython 的自动完成功能进行查看。

```
>>> ma.masked_ # 按 Tab 键进行自动完成
ma.masked_all           ma.masked_inside        ma.masked_outside
ma.masked_all_like      ma.masked_invalid       ma.masked_print_option
ma.masked_array         ma.masked_less          ma.masked_singleton
ma.masked_equal         ma.masked_less_equal    ma.masked_values
ma.masked_greater       ma.masked_not_equal     ma.masked_where
ma.masked_greater_equal ma.masked_object
```

通过函数名可以很容易猜出它们的功能，这里就不一一详细叙述了，下面我们看一个具体的例子。

下面的程序先创建一个形状为(3,10000)、值为正态分布的数组 x。然后调用 masked_outside()创建一个掩码数组，将其中小于 0 或大于 1 的值设置为无效，然后计算掩码数组第 1 轴上各个数值的方差：

```
>>> x = np.random.normal(size=(3,10000))
>>> mx = ma.masked_outside(x, 0, 1)
>>> mx.var(axis=1)
masked_array(data = [ 0.07932837  0.08013028  0.08050745],
             mask = False,
       fill_value = 1e+20)
```

而原始数组的方差为：

```
>>> x.var(axis=1)
array([ 1.03902508,  0.99990538,  1.00297267])
```

2.7 文件存取

NumPy 提供了多种存取数组内容的文件操作函数。保存数组数据的文件可以是二进制格式或文本格式。二进制格式的文件又分为 NumPy 专用的格式化二进制类型和无格式类型。

使用数组对象的 tofile()方法可以方便地将数组中的数据以二进制格式写进文件。tofile()输出的数据不保存数组形状和元素类型等信息。因此用 fromfile()函数读回数据时需要用户指定元素类型，并对数组的形状进行适当的修改：

```
>>> a = np.arange(0,12)
>>> a.shape = 3,4
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.tofile("a.bin")
>>> b = np.fromfile("a.bin", dtype=np.float) # 按照 float 类型读入数据
>>> b # 读入的数据是错误的
array([  2.12199579e-314,   6.36598737e-314,   1.06099790e-313,
         1.48539705e-313,   1.90979621e-313,   2.33419537e-313])
>>> a.dtype # 查看 a 的 dtype
dtype('int32')
>>> b = np.fromfile("a.bin", dtype=np.int32) # 按照 int32 类型读入数据
>>> b # 数据是一维的
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> b.shape = 3, 4 # 按照数组 a 的 shape 修改数组 b 的 shape
>>> b # 这次终于正确了
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

从上面的例子可以看出，在读入数据时需要正确设置 dtype 参数，并修改数组的 shape 属性才能得到和原始数据一致的结果。无论数据的排列顺序是 C 语言格式还是 Fortran 语言格式，tofile()都统一使用 C 语言格式输出。此外如果指定了 sep 参数，fromfile()和 tofile()将以文本格式对数组进行输入输出。sep 参数指定的是文本数据中数值的分隔符。

load()和 save()用 NumPy 专用的二进制格式保存数据，它们会自动处理元素类型和形状等信息：

```
>>> np.save("a.npy", a)
>>> c = np.load( "a.npy" )
>>> c
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

如果想将多个数组保存到一个文件中，可以使用 savez()。savez()的第一个参数是文件名，其后的参数都是需要保存的数组，也可以使用关键字参数为数组起名，非关键字参数传递的数组会自动起名为 arr_0、arr_1、…。savez()输出的是一个扩展名为 npz 的压缩文件，其中每个文件都是一个用 save()保存的 npy 文件，文件名和数组名相同。load()自动识别 npz 文件，并且返回一个类似于字典的对象，可以通过以数组名作为键来获取数组的内容：

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> b = np.arange(0, 1.0, 0.1)
>>> c = np.sin(b)
>>> np.savez("result.npz", a, b, sin_array = c)
>>> r = np.load("result.npz")
>>> r["arr_0"] # 数组 a
array([[1, 2, 3],
       [4, 5, 6]])
>>> r["arr_1"] # 数组 b
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
>>> r["sin_array"] # 数组 c
array([ 0.        ,  0.09983342,  0.19866933,  0.29552021,  0.38941834,
        0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691])
```

用解压软件打开“result.npz”文件，会发现其中有三个文件：“arr_0.npy”、“arr_1.npy”和“sin_array.npy”，其中分别保存着数组 a、b、c 的内容。

save()和 savez()输出的二进制文件有特殊的格式，较难用其他语言编写的程序读入。

savetxt()和 loadtxt()可以读写保存一维和二维数组的文本文件。例如可以用它们读写 CSV 格式的文本文件：

```
>>> a = np.arange(0,12,0.5).reshape(4,-1)
>>> np.savetxt("a.txt", a) # 默认按照'%.18e'格式保存数值，以空格分隔
>>> np.loadtxt("a.txt")
array([[  0. ,   0.5,   1. ,   1.5,   2. ,   2.5],
       [  3. ,   3.5,   4. ,   4.5,   5. ,   5.5],
       [  6. ,   6.5,   7. ,   7.5,   8. ,   8.5],
       [  9. ,   9.5,  10. ,  10.5,  11. ,  11.5]])
```

```
>>> np.savetxt("a.txt", a, fmt="%d", delimiter=",") #改成保存为整数，以逗号分隔
>>> np.loadtxt("a.txt",delimiter=",") # 读入的时候也需要指定逗号分隔
array([[  0.,   0.,   1.,   1.,   2.,   2.],
       [  3.,   3.,   4.,   4.,   5.,   5.],
       [  6.,   6.,   7.,   7.,   8.,   8.],
       [  9.,   9.,  10.,  11.,  11.,  11.]])
```

有的 CSV 文件中除了保存数值之外，还保存一些说明文字，例如第一行和第一列通常为列名和行名。如果需要忽略 CSV 文件的第一行和第一列，可以先将文件读为字符串数组，然后取出需要的部分，转换为数值数组。例如对于下面的 CSV 数据文件：

```
姓名,年龄,体重,身高
张三,30,75,165
李四,45,60,170
王五,15,30,120
```

可以采用如下程序读入其中的数值部分：



numpy_read_csv.py
读取 CSV 文件

```
>>> tmp = np.loadtxt("test.csv", dtype=np.str, delimiter=",")
>>> data = tmp[1:,1:].astype(np.float)
>>> data
array([[  30.,   75.,  165.],
       [  45.,   60.,  170.],
       [  15.,   30.,  120.]])
```

此外，使用结构数组也能读入这样的文件，并且可以使用不同的元素类型保存每个列的值，下面先定义结构数组的类型：

```
>>> persontype = np.dtype({
...     'names':['name', 'age', 'weight', 'height'],
...     'formats':['S32','i', 'f', 'f']})
```

由于文件中的第一行不是数据，因此需要先打开数据文件，读取了第一行之后，再把文件对象传递给 loadtxt()：

```
>>> f = file("test.csv")
>>> f.readline()
>>> data = np.loadtxt(f, delimiter=",", dtype=persontype)
>>> print data
[('\xe5\xbc\xa0\xe4\xb8\x89', 30, 75.0, 165.0)
```

```
    ('\xe6\x9d\x8e\xe5\x9b\x9b', 45, 60.0, 170.0)
    ('\xe7\x8e\x8b\xe4\xba\x94', 15, 30.0, 120.0)]
```

实际上，前面介绍的所有读写文件的函数都可以直接使用已经打开的文件对象。如果使用文件对象，可以将多个数组存储到一个 npy 文件中：

```
>>> a = np.arange(8)
>>> b = np.add.accumulate(a)
>>> c = a + b
>>> f = file("result.npy", "wb")
>>> np.save(f, a) # 顺序将数组 a、b、c 保存到文件对象 f 中
>>> np.save(f, b)
>>> np.save(f, c)
>>> f.close()
>>> f = file("result.npy", "rb")
>>> np.load(f) # 顺序从文件对象 f 中读取内容
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> np.load(f)
array([ 0,  1,  3,  6, 10, 15, 21, 28])
>>> np.load(f)
array([ 0,  2,  5,  9, 14, 20, 27, 35])
```

2.8　内存映射数组

当需要存取一个很大的数据文件中的一小部分数据时，将整个文件读入内存进行操作显然很费资源的。使用内存映射数组(memory-mapped-file array)能够帮助我们快速解决问题。



在 32 位操作系统中，程序使用的有内存映射数组的空间总和小于 2GB。

内存映射数组使用 memmap()创建，它的调用参数如下：

```
memmap(filename, dtype=uint8, mode="r+", offset=0, shape=None, order="C")
```

- filename：存储数组的文件名，除了"w+"模式之外，文件必须存在，并且已经存储有数据。
- dtype：NumPy 的数据类型，可以是内置的数据类型，也可以是用户自己定义的结构类型。
- offset：文件中存储数据的起始位置，以字节为单位。

- mode：文件操作模式，"r"表示只读，"c"表示可以修改数组但不写入文件，"r+"表示可对数组进行读写，并自动将结果保存回文件，"w+"表示创建文件或覆盖已有的文件。
- order：元素排列格式，"C"表示 C 语言格式，"F"表示 Fortran 语言格式。

下面通过一个实例说明内存映射数组的用法。首先使用"w+"模式创建一个内存映射数组，运行完下面的语句之后，将会生成一个名为“tmp.dat”的文件，内容是 10 个值为 0 的字节。

```
>>> a = np.memmap("tmp.dat", mode="w+", shape=(2,5))
>>> a
memmap([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
>>> file("tmp.dat").read()
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

接下来将数组 a 中每个元素都改为字符'a'的 ASCII 码，然后调用 flush()方法将所有改动写入文件中：

```
>>> a[:] = ord("a")
>>> a.flush()
>>> file("tmp.dat").read()
'aaaaaaaaaa'
```

这时“tmp.dat”文件的内容变成了 10 个字符'a'。然后用"c"模式打开此文件，修改其中的一些元素为字符'b'，虽然内存中的数据已经修改，但是即使调用 flush()，也不会将结果写入文件中：

```
>>> b = np.memmap("tmp.dat", mode="c", shape=(2,5))
>>> b
memmap([[97, 97, 97, 97, 97],
       [97, 97, 97, 97, 97]], dtype=uint8)
>>> b[1] = ord("b")
>>> b
memmap([[97, 97, 97, 97, 97],
       [98, 98, 98, 98, 98]], dtype=uint8)
>>> b.flush()
>>> file("tmp.dat").read()
'aaaaaaaaaa'
```

“tmp.dat”文件的内容仍然是 10 个'a'。如果使用"r+"模式，就会将改动写入文件中：

```
>>> c = np.memmap("tmp.dat", shape=(2,5))
>>> c[1] = ord("c")
>>> c.flush()
>>> file("tmp.dat").read()
```

```
'aaaaaccccc'
```

此时再看看数组 a 和 b 中的值，数组 a 的值改变了，而数组 b 没有改变：

```
>>> a
memmap([[97, 97, 97, 97, 97],
       [99, 99, 99, 99, 99]], dtype=uint8)
>>> b
memmap([[97, 97, 97, 97, 97],
       [98, 98, 98, 98, 98]], dtype=uint8)
```

许多格式的文件(例如 WAV 和 BMP)都有一个文件头，然后才是数据内容。为了正确读取数据，需要使用 offset 参数指定数据的偏移量，下面的例子演示了如何使用 memmap 读写 BMP 文件。读者可以使用 Windows 自带的画笔软件手工创建一张 1000*1000 的 24 bit 的 bmp 文件，也可通过运行本书提供的程序来创建。



create_bmp.py
创建一个大小为 1000*1000 的 24 bit 的 BMP 文件

这里我们不关心文件头中有些什么内容，只想知道 BMP 文件头的大小。查看创建的 BMP 文件的大小，为 3 000 054 个字节，显然 BMP 文件头的大小为 54 个字节，其后是 1000*1000*3 个字节，用来表示图像中每个像素的颜色。

然后使用下面的程序为图像中的每个像素赋值：



numpy_change_bmp.py
使用 memmap 数组修改 BMP 文件中像素点的颜色

```
import numpy as np

# shape = 高，宽，颜色
bmp = np.memmap("tmp.bmp", offset=54, shape=(1000,1000,3)) ❶

# 产生一组从 0 到 255 的渐变值
tmp = np.linspace(0, 255, 1000).astype(np.uint8) ❷

# 水平蓝色渐变
bmp[:, :, 0] = tmp  ❸
# 垂直绿色渐变
bmp[:, :, 1] = tmp.reshape(-1,1)  ❸
# 红色成分
bmp[:, :, 2] = 127  ❹

bmp.flush() ❺
```

❶调用 np.memmap()来创建内存映射数组。offset 参数指定了文件中数据的偏移量，而 shape 参数指定了数据的形状。BMP 文件的高度和宽度均为 1000 个像素，每个像素点为 3 个 uint8 的整数，因此得到的数组 bmp 是一个三维数组，其形状为(1000,1000,3)。第 0 轴表示 BMP 图像的高，第 1 轴表示宽，第 2 轴表示每个像素点的蓝绿红三种颜色的分量。

❷使用 linspace()产生一组从 0 到 255 渐变的 uint8 数组，❸并将其分别赋值给 bmp 数组的蓝色和绿色部分，❹红色部分我们全部设置为 127。❺最后调用 flush()，确保内存中的数据能写回 BMP 文件中。这样就得到了一张彩色的渐变图像。

第 3 章

SciPy——数值计算库

SciPy 在 NumPy 的基础上增加了众多的数学、科学以及工程计算中常用的模块，例如线性代数、常微分方程数值求解、信号处理、图像处理、稀疏矩阵，等等。在本章，将通过实例介绍 SciPy 中常用的一些模块。为了方便读者理解，在实例程序中会使用 matplotlib 和 Mayavi 绘制二维和三维图表，在阅读实例源程序时，读者可以忽略绘图部分，在后续章节中，我们会对这些绘图库进行详细介绍。

3.1 常数和特殊函数

SciPy 的 constants 模块包含了众多的物理常数：

```
>>> from scipy import constants as C
>>> C.c # 真空中的光速
299792458.0
>>> C.h # 普朗克常数
6.6260693000000002e-34
```

在字典 physical_constants 中，以物理常量名为键，对应的值是一个含有三个元素的元组，分别为常数值、单位及误差，例如下面的程序可以查看电子质量：

```
>>> C.physical_constants["electron mass"]
(9.1093825999999998e-31, 'kg', 1.5999999999999999e-37)
```

除了物理常数之外，constants 模块中还包括许多单位信息，例如：

```
>>> C.mile # 1 英里等于多少米
1609.3439999999998
>>> C.inch # 1 英寸等于多少米
0.025399999999999999
>>> C.gram # 1 克等于多少千克
0.001
>>> C.pound # 1 磅等于多少千克
0.45359236999999997
```

SciPy 的 special 模块是一个非常完整的函数库，其中包含了基本数学函数、特殊数学函数以及 NumPy 中出现的所有函数。由于函数数量众多，本节仅对其进行简要介绍。至于具体包含的函数列表，请读者参考 SciPy 的帮助文档。

伽玛函数 Γ 是概率统计学中经常出现的一个特殊函数，它的计算公式如下：

$$\Gamma(z)=\int_0^\infty t^{z-1}e^{-t}dt$$

显然，通过此公式计算 Γ 函数的值是比较麻烦的，我们可以用 special 模块中的 gamma() 进行计算：

```
>>> import scipy.special as S
>>> S.gamma(4)
6.0
>>> S.gamma(0.5)
1.7724538509055159
>>> S.gamma(1+1j) # gamma 函数支持复数
(0.49801566811835629-0.15494982830181106j)
>>> S.gamma(1000)
inf
```

Γ 函数是阶乘函数在实数和复数范围上的扩展，它的增长速度非常快，因为 1000 的阶乘已经超过了双精度浮点数的表示范围，因此结果是无穷大。为了计算更大的范围，可以使用 gammaln()：

```
>>> gammaln(1000)
5905.2204232091817
```

gammaln(x)计算 $\ln(|\Gamma(x)|)$的值，它使用特殊的算法，直接计算 Γ 函数的对数值，因此可以表示更大的范围。

special 模块中的某些函数并不是数学意义上的特殊函数，例如 log1p(x)计算 log(1+x)的值。这是由于浮点数的精度有限，无法很精确地表示十分接近 1 的实数。例如无法用浮点数表示“1 + 1e-20”的值，因此“log(1+1e-20)”的值为 0，而当使用 log1p()时，则可以很精确地计算：

```
>>> 1 + 1e-20
1.0
>>> S.log(1+1e-20)
0.0
>>> S.log1p(1e-20)
9.9999999999999995e-21
```

实际上当 x 非常小时，log1p(x)约等于 x，这可以通过对 log(1+x)进行泰勒级数展开加以证明。在下一章我们会学习如何用符号计算库 SymPy 进行泰勒级数展开。

3.2 优化——optimize

SciPy 的 optimize 模块提供了许多数值优化算法，本节对其中的数据拟合、函数最小值以及解非线性方程组等进行简单的介绍。

3.2.1 最小二乘拟合

假设有一组实验数据(x_i, y_i)，我们事先知道它们之间应该满足某函数关系 $y_i=f(x_i)$，通过这些已知信息，需要确定函数 f 的一些参数。例如，如果函数 f 是线性函数 $f(x)=kx+b$，那么参数 k 和 b 就是需要确定的值。

如果用 p 表示函数中需要确定的参数，那么目标就是找到一组 p，使得下面的函数 S 的值最小：

$$S(p)=\sum_{i=1}^{M}[y_i-f(x_i,p)]^2$$

这种算法被称为最小二乘拟合(Least-square fitting)。在 optimize 模块中，可以使用 leastsq()对数据进行最小二乘拟合计算。leastsq()的用法很简单，只需要将计算误差的函数和待确定参数的初始值传递给它即可。下面是用 leastsq()对线性函数进行拟合的程序。



scipy_least_square_line.py
用最小二乘法拟合直线，并显示误差曲面

```
import numpy as np
from scipy.optimize import leastsq

X = np.array([ 8.19,  2.72,  6.39,  8.71,  4.7 ,  2.66,  3.78])
Y = np.array([ 7.01,  2.78,  6.47,  6.71,  4.1 ,  4.23,  4.05])

def residuals(p): ❶
    "计算以 p 为参数的直线和原始数据之间的误差"
    k, b = p
    return Y - (k*X + b)

# leastsq 使得 residuals()的输出数组的平方和最小，参数的初始值为[1,0]
r = leastsq(residuals, [1, 0]) ❷
k, b = r[0]
print "k =",k, "b =",b
```

程序的输出为：

```
k = 0.613495346194 b = 1.79409255555
```

图 3-1(左)直观地显示了原始数据、拟合直线以及它们之间的误差。❶residuals()的参数 p 是拟合直线的参数，函数返回的是原始数据和拟合直线之间的误差。图中用数据点到拟合直线在 Y 轴上的距离表示误差。❷leastsq()使得这些误差的平方和最小，即图中所有正方形的面积之和最小。

由前面的函数 S 的公式可知，对于直线拟合来说，误差的平方和是直线参数 k 和 b 的二次多项式函数，因此可以用图 3-1(右)所示的曲面直观地显示误差平方和与两个参数之间的关系。图中用灰色圆球表示曲面的最小点，它的 X-Y 轴的坐标就是 leastsq()的拟合结果。下面的函数 S()用来计算误差曲面，其参数 k 和 b 均为二维数组：

```
def S(k, b):
    "计算直线 y=k*x+b 和原始数据 X、Y 的误差的平方和"
    error = np.zeros(k.shape)
    for x, y in zip(X, Y):
        error += (y - (k*x + b))**2
    return error
```

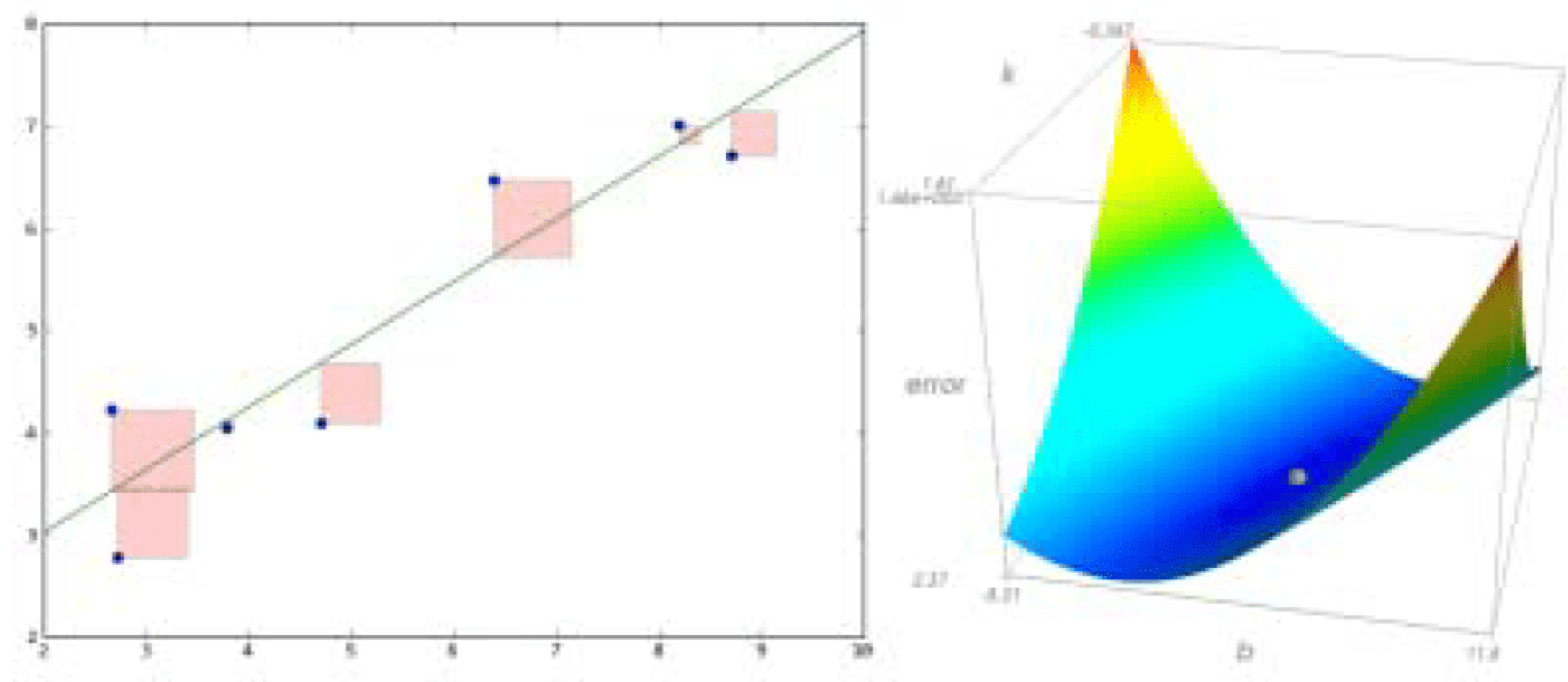


图 3-1　最小二乘法拟合的直线以及误差(左)，误差和两个参数 k、b 之间的关系(右)

接下来，让我们再看一个对正弦波数据进行拟合的例子：

scipy_least_square_sin.py
使用最小二乘法对带噪声的正弦波数据进行拟合

```
def func(x, p): ❶
    """
    数据拟合所用的函数: A*sin(2*pi*k*x + theta)
    """
```

```
    A, k, theta = p
    return A*np.sin(2*np.pi*k*x+theta)

def residuals(p, y, x): ❷
    """
    实验数据 x、y 和拟合函数之间的差，p 为拟合需要找到的系数
    """
    return y - func(x, p)

x = np.linspace(0, 2*np.pi, 100)
A, k, theta = 10, 0.34, np.pi/6 # 真实数据的函数参数
y0 = func(x, [A, k, theta]) # 真实数据
# 加入噪声之后的实验数据
y1 = y0 + 2 * np.random.randn(len(x)) ❸

p0 = [7, 0.2, 0] # 第一次猜测的函数拟合参数

# 调用 leastsq 进行数据拟合
# residuals 为计算误差的函数
# p0 为拟合参数的初始值
# args 为需要拟合的实验数据
plsq = leastsq(residuals, p0, args=(y1, x)) ❹

print u"真实参数:", [A, k, theta]
print u"拟合参数", plsq[0] # 实验数据拟合后的参数
```

程序中，❶要拟合的目标函数 func()是一个正弦函数，它的参数 p 是一个数组，包含决定正弦波的三个参数：A、k、theta，分别对应正弦函数的振幅、频率和相角。❸待拟合的实验数据是一组包含噪声的数据：(x, y1)，其中数组 y1 在标准正弦波数据 y0 之上添加了随机噪声。

❹用 leastsq()对带噪声的实验数据(x, y1)进行数据拟合，它可以找到数组 x 和真实数据 y0 之间的正弦关系，即确定 A、k、theta 等参数。和前面的直线拟合程序不同的是，这里我们将(y1, x)传递给 args 参数。leastsq()会将这两个额外的参数传递给 residuals()。❷因此 residuals()有三个参数，p 是正弦函数的参数，y 和 x 是表示实验数据的数组。下面是程序的输出：

```
真实参数: [10, 0.34000000000000002, 0.52359877559829882]
拟合参数: [-9.84152775  0.33829767 -2.68899335]
```

我们看到：拟合结果中振幅和频率基本一致，但相角却完全不同。由于正弦函数具有周期性，这两个相角实际上相差整数个周期。如图 3-2 所示，拟合的结果和实际的数据是一致的(见封二彩插)。

如果频率的初值和真实值差别很大，拟合结果中的频率参数可能不能收敛于实际的频率。这时可以通过其他方法先估算一个频率的近似值。

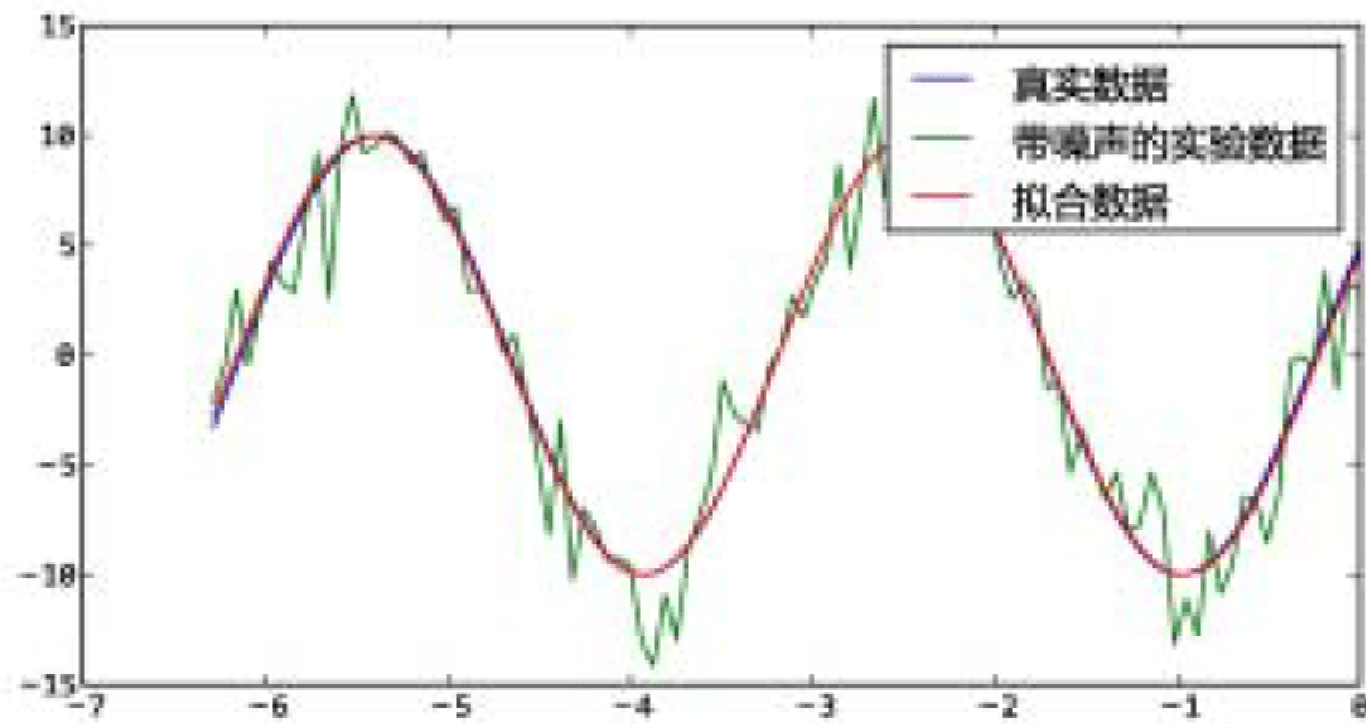


图 3-2 使用最小二乘法对带噪声的正弦波数据进行拟合

3.2.2 函数最小值

optimize 模块还提供了许多求函数最小值的算法：fmin、fmin_powell、fmin_cg、fmin_bfgs 等。下面我们用一个实例观察这些“fmin*()”是如何找到函数的最小值的。在本例中，要计算最小值的函数 $f(x,y)$ 为：

$$f(x,y)=(1-x)^2+100(y-x^2)^2$$

为了提高运算速度和精度，有些“fmin*()”带有一个 fprime 参数，它是计算目标函数 f 对各个自变量的偏导数的函数。$f(x,y)$ 对变量 x 和 y 的偏导函数为：

$$\frac{\partial f}{\partial x}=-2+2x-400x(y-x^2),\frac{\partial f}{\partial y}=200y-200x^2$$

这个函数叫做 Rosenbrock 函数，它经常用来测试最小化算法的收敛速度。它有一个十分平坦的山谷区域，收敛到此山谷区域比较容易，但是在山谷区域搜索到最小点则比较困难。根据函数的计算公式不难看出此函数的最小值是 0，在(1,1)处。

下面的程序计算 $f(x, y)$ 的最小值，并且绘制出 $f(x, y)$ 所表示的曲面和寻找最小值时的搜索路径。

scipy_fmin_demo.py
观察 fmin*函数计算最小值时的路径

```
import scipy.optimize as opt
import numpy as np
import sys

points = []
```

```
def f(p): ❶
    x, y = p
    z = (1-x)**2 + 100*(y-x**2)**2
    points.append((x,y,z))
    return z

def fprime(p): ❷
    x, y = p
    dx = -2 + 2*x - 400*x*(y - x**2)
    dy = 200*y - 200*x**2
    return np.array([dx, dy])

init_point =(-2,-2) ❸

try:
    method = sys.argv[1]
except:
    method = "fmin_bfgs"

fmin_func = opt.__dict__[method] ❹
if method in ["fmin", "fmin_powell"]:
    result = fmin_func(f, init_point)     #参数为目标函数和初始值
elif method in ["fmin_cg", "fmin_bfgs", "fmin_l_bfgs_b", "fmin_tnc"]:
    result = fmin_func(f, init_point, fprime)     #参数为目标函数、初始值和导函数
elif method in ["fmin_cobyla"]:
    result = fmin_func(f, init_point, [])
else:
    print "fmin function not found"
    sys.exit(0)

### 绘图部分
import pylab as pl
p = np.array(points)
xmin, xmax = np.min(p[:,0])-1, np.max(p[:,0])+1
ymin, ymax = np.min(p[:,1])-1, np.max(p[:,1])+1
Y, X = np.ogrid[ymin:ymax:500j,xmin:xmax:500j]
Z = np.log10(f((X, Y)))
zmin, zmax = np.min(Z), np.max(Z)
pl.imshow(Z, extent=(xmin,xmax,ymin,ymax), origin="bottom", aspect="auto")
pl.plot(p[:,0], p[:,1])
pl.scatter(p[:,0], p[:,1],c=range(len(p)))
pl.xlim(xmin,xmax)
pl.ylim(ymin,ymax)
pl.show()
```

❶f()计算 $f(x,y)$的函数值，为了记录下最小化过程中的计算轨迹，在 f()中将每个计算过的点都添加进全局列表 points 中。❷fprime()计算 $f(x,y)$对两个自变量在 p 处的偏导函数的值。

❸最小化的初值设置为(-2,2)，❹此程序从 optimize 模块的__dict__字典中获得由命令行参数指定的最小值函数。不同的"fmin*()"参数有所不同，例如有些算法不需要 fprime()。

最后将曲面和计算点的路径绘制成图，效果如图 3-3 所示(见封二彩插)。它显示了 fmin()和 fmin_bfgs()搜索最小值时的轨迹。可以看出：由于 fmin_bfgs()使用了偏导函数 fprime()，因此它能够更快地找到最小值。这里使用图像表示二维函数的值，值越大则颜色越红，值越小则颜色越蓝。为了更清晰地显示函数的山谷区域，图中显示的实际上是通过对数函数 log10()对 $f(x,y)$进行处理之后的结果。

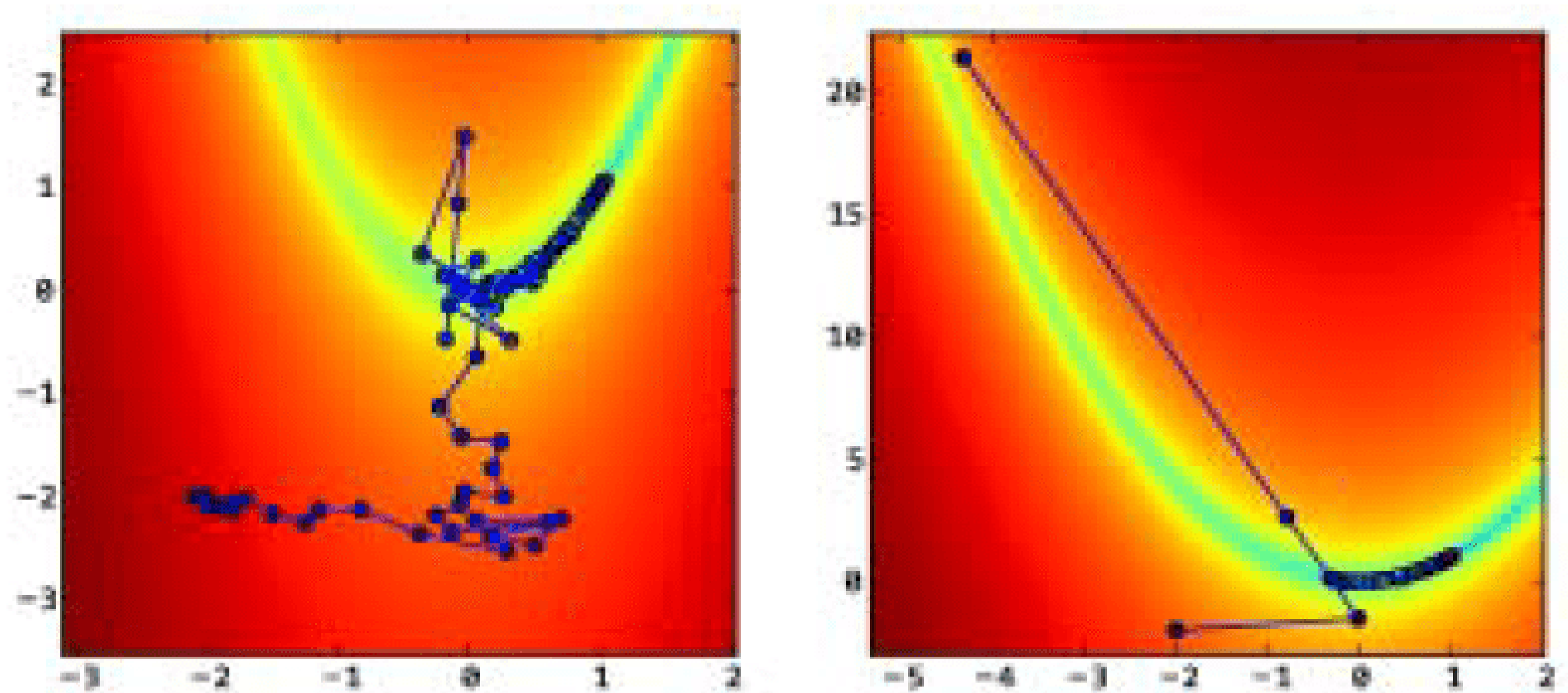


图 3-3　fmin 最小化时的计算轨迹(左)，fmin_bfgs 最小化时的计算轨迹(右)

3.2.3　非线性方程组求解

optimize 模块中的 fsolve()可以对非线性方程组进行求解，它的基本调用形式如下：

```
fsolve(func, x0)
```

func 是计算方程组误差的函数，它的参数 x 是一个数组，其值为方程组的一组可能的解。func 返回将 x 代入方程组之后得到的每个方程的误差，x0 为未知数的一组初始值。假设要对下面的方程组进行求解：

$$f1(u1,u2,u3)=0,\quad f2(u1,u2,u3)=0,\quad f3(u1,u2,u3)=0$$

那么 func 可以如下定义：

```
def func(x):
    u1,u2,u3 = x
    return [f1(u1,u2,u3), f2(u1,u2,u3), f3(u1,u2,u3)]
```

下面我们看一个对下列方程组求解的例子。

$$5x_1+3=0，4x_0^2-2sin(x_1 x_2)=0，x_1 x_2-1.5=0$$



scipy_fsolve.py
使用 fsolve 求非线性方程组的解

```
from scipy.optimize import fsolve
from math import sin

def f(x): ❶
    x0, x1, x2 = x.tolist() ❷
    return [
        5*x1+3,
        4*x0*x0 - 2*sin(x1*x2),
        x1*x2 - 1.5
    ]

# f 计算方程组的误差，[1,1,1]是未知数的初始值
result = fsolve(f, [1,1,1]) ❸
print result
print f(result)
```

程序的输出为：

```
[-0.70622057 -0.6        -2.5       ]
[0.0, -9.1260332624187868e-14, 5.3290705182007514e-15]
```

❶f()是计算方程组误差的函数，x 参数是一组可能的解。fsolve()在调用 f()时，传递给 f()的参数是一个数组。❷先调用数组的 tolist()方法，将其转换为 Python 的标准浮点数列表，然后调用 math 模块中的函数进行运算。因为在进行单个数值的运算时，标准浮点类型比 NumPy 的浮点类型要快许多，所以把数值都转换成标准浮点数类型，能缩短一些计算时间。❸调用 fsolve()时，传递计算误差的函数 f()以及未知数的初始值。

在对方程组进行求解时，fsolve()会自动计算方程组在某点对各个未知数变量的偏导数，这些偏导数组成一个二维数组，数学上称之为雅可比矩阵。如果方程组中的未知数很多，而与每个方程有关联的未知数较少，即雅可比矩阵比较稀疏时，将计算雅可比矩阵的函数作为参数传递给 fsolve()，将能大幅度提高运算速度。笔者在一个模拟计算的程序中需要求解有 50 个未知数的非线性方程组。每个方程平均与 6 个未知数相关，通过传递计算雅可比矩阵的函数使 fsolve()的计算速度提高了 4 倍。

雅可比矩阵

雅可比矩阵是一阶偏导数以一定方式排列的矩阵，它给出了可微分方程与给定点的最

优线性逼近，因此类似于多元函数的导数。例如前面的函数 f1、f2、f3 和未知数 u1、u2、u3 的雅可比矩阵如下：

$$\begin{bmatrix} \frac{\partial f1}{\partial u1} & \frac{\partial f1}{\partial u2} & \frac{\partial f1}{\partial u3} \\ \frac{\partial f2}{\partial u1} & \frac{\partial f2}{\partial u2} & \frac{\partial f2}{\partial u3} \\ \frac{\partial f3}{\partial u1} & \frac{\partial f3}{\partial u2} & \frac{\partial f3}{\partial u3} \end{bmatrix}$$

使用雅可比矩阵的求解程序如下：



scipy_fsolve_jacobian.py
使用雅可比行列式求非线性方程组的解

```
def j(x): ❶
    x0, x1, x2 = x.tolist()
    return [
        [0, 5, 0],
        [8*x0, -2*x2*cos(x1*x2), -2*x1*cos(x1*x2)],
        [0, x2, x1]
    ]

result = fsolve(f, [1,1,1], fprime=j) ❷
```

❶计算雅可比矩阵的函数 j()和 f()一样，其 x 参数是未知数的一组值，它计算非线性方程组在 x 处的雅可比矩阵。❷通过 fprime 参数将 j()传递给 fsolve()。由于本例中的未知数很少，因此计算雅可比矩阵并不能显著地提高计算速度。

3.3 插值——interpolate

插值是一种通过已知的离散数据来求未知数据的方法。与拟合不同的是，它要求曲线通过所有的已知数据。SciPy 的 interpolate 模块提供了许多对数据进行插值运算的函数。

3.3.1 B 样条曲线插值

一维数据的插值运算可以通过 interp1d()完成。其调用形式如下，它实际上不是函数而是一个类：

```
interp1d(x, y, kind='linear', ...)
```

其中：参数 x 和 y 是一系列已知的数据点；参数 kind 是插值类型，可以是字符串或整数。参数 kind 给出了插值的 B 样条曲线的阶数，可以有如下候选值：

- 'zero'、'nearest'：阶梯插值，相当于 0 阶 B 样条曲线。
- 'slinear'、'linear'：线性插值，用一条直线连接所有的取样点，相当于 1 阶 B 样条曲线，'slinear'使用扩展库中的相关函数进行计算，而'linear'则直接使用 Python 编写的函数进行运算，它们的结果一样。
- 'quadratic'、'cubic'：2 阶和 3 阶 B 样条曲线，更高阶的曲线可以直接使用整数值指定。
- interp1d 对象可以计算 x 的取值范围之内任意点的函数值。它可以像函数一样直接调用，像 NumPy 的 ufunc 函数一样能对数组中的每个元素进行计算，并返回一个新的数组。

下面的程序演示了参数 kind 与其对应的插值曲线，其结果如图 3-4 所示(见封二彩插)。



scipy_interp1d.py
使用 interp1d()对数据进行各阶插值

```
import numpy as np
from scipy import interpolate
import pylab as pl

x = np.linspace(0, 10, 11)
y = np.sin(x)

xnew = np.linspace(0, 10, 101)
pl.plot(x,y,'ro')
for kind in ['nearest', 'zero', 'slinear', 'quadratic']:
    f = interpolate.interp1d(x,y,kind=kind) ❶
    ynew = f(xnew) ❷
    pl.plot(xnew, ynew, label=str(kind))

pl.legend(loc='lower right')
pl.show()
```

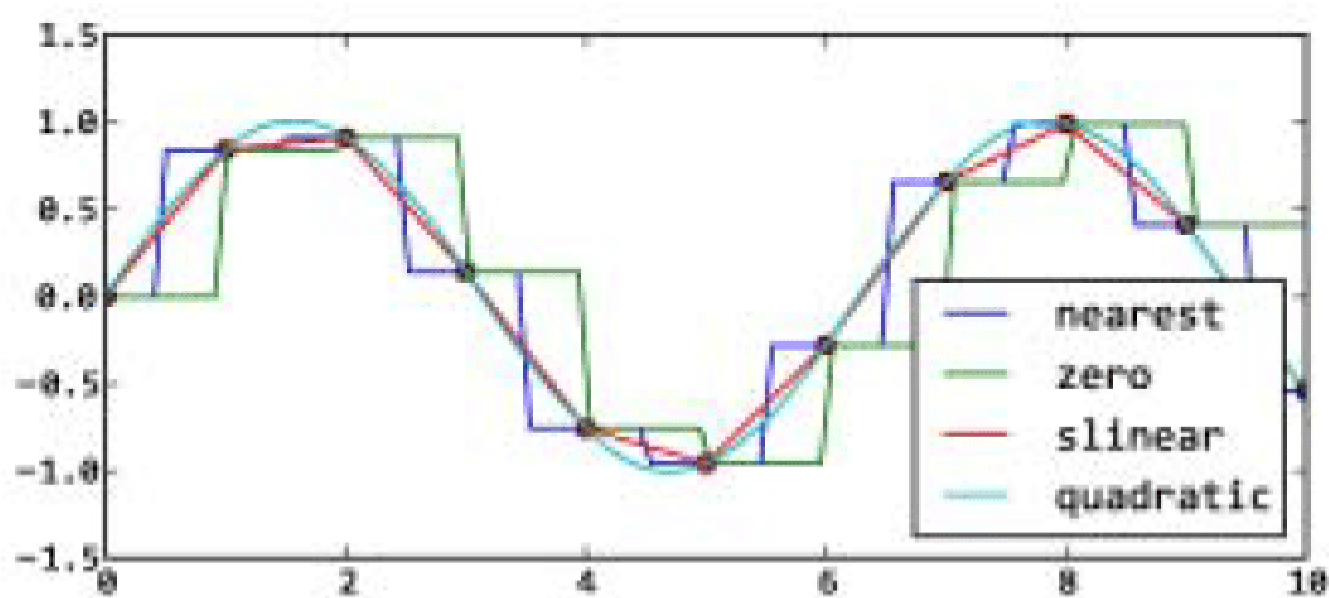


图 3-4　使用参数 kind 指定插值曲线的阶数

程序中我们使用循环对相同的数据进行 4 种不同阶数的插值运算。❶首先使用数据点创建一个 interp1d 对象 f，通过 kind 参数指定其阶数。❷调用 f()计算出一系列的插值结果。本例中，决定插值曲线的数据点一共有 11 个，插值之后的曲线数据点有 101 个。

3.3.2 外推和 Spline 拟合

前面介绍的 interp1d 类要求其参数 x 是一个递增序列，并且只能在 x 的取值范围之内进行内插计算，不能用它进行外推运算，即无法计算 x 的取值范围之外的数据点。UnivariateSpline 类的插值运算比 interp1d 更高级，它支持外推运算，其调用形式如下：

```
UnivariateSpline(x, y, w=None, bbox=[None, None], k=3, s=None)
```


- x、y 是保存数据点的 X-Y 坐标的数组，其中 x 必须是递增序列。
- w 是为每个数据点指定的权重值。
- k 为样条曲线的阶数。
- s 是平滑参数，它使得最终生成的样条曲线满足条件 $\sum(w\cdot(y-spline(x)))^2 \le s$，即当 s>0 时，样条曲线并不一定通过各个数据点。为了让曲线通过所有数据点，必须将 s 参数设置为 0。

下面的程序演示了如何使用 UnivariateSpline 对数据进行插值、外推及样条曲线拟合：



scipy_uspline.py
使用 UnivariateSpline 进行插值运算

```
x1 = np.linspace(0, 10, 20)
y1 = np.sin(x1)
sx1 = np.linspace(0, 12, 100)
sy1 = interpolate.UnivariateSpline(x1, y1, s=0)(sx1) ❶

x2 = np.linspace(0, 20, 200)
y2 = np.sin(x2) + np.random.standard_normal(len(x2))*0.2
sx2 = np.linspace(0, 20, 2000)
sy2 = interpolate.UnivariateSpline(x2, y2, s=8)(sx2) ❷
```

❶如图 3-5(上)所示，UnivariateSpline 能够进行外推运算，虽然输入数据中没有 X 轴大于 10 的点，但是它能计算出 X 轴在 0 到 12 的插值结果(见封二彩插)。在 X 轴大于 10 的部分，样条曲线仍然呈现出和正弦波类似的形状，越远离输入数据范围，误差会越大，因此外推的范围是有限的。由于 s 参数为 0，因此插值曲线经过所有的数据点。

❷图 3-5(下)则显示了 s 参数不为零时的结果，对于带噪声的输入数据，选择合适的 s 参数能够使得样条曲线接近无噪声时的波形，可以把它看作使用样条曲线对数据进行拟合运算(见封二彩插)。

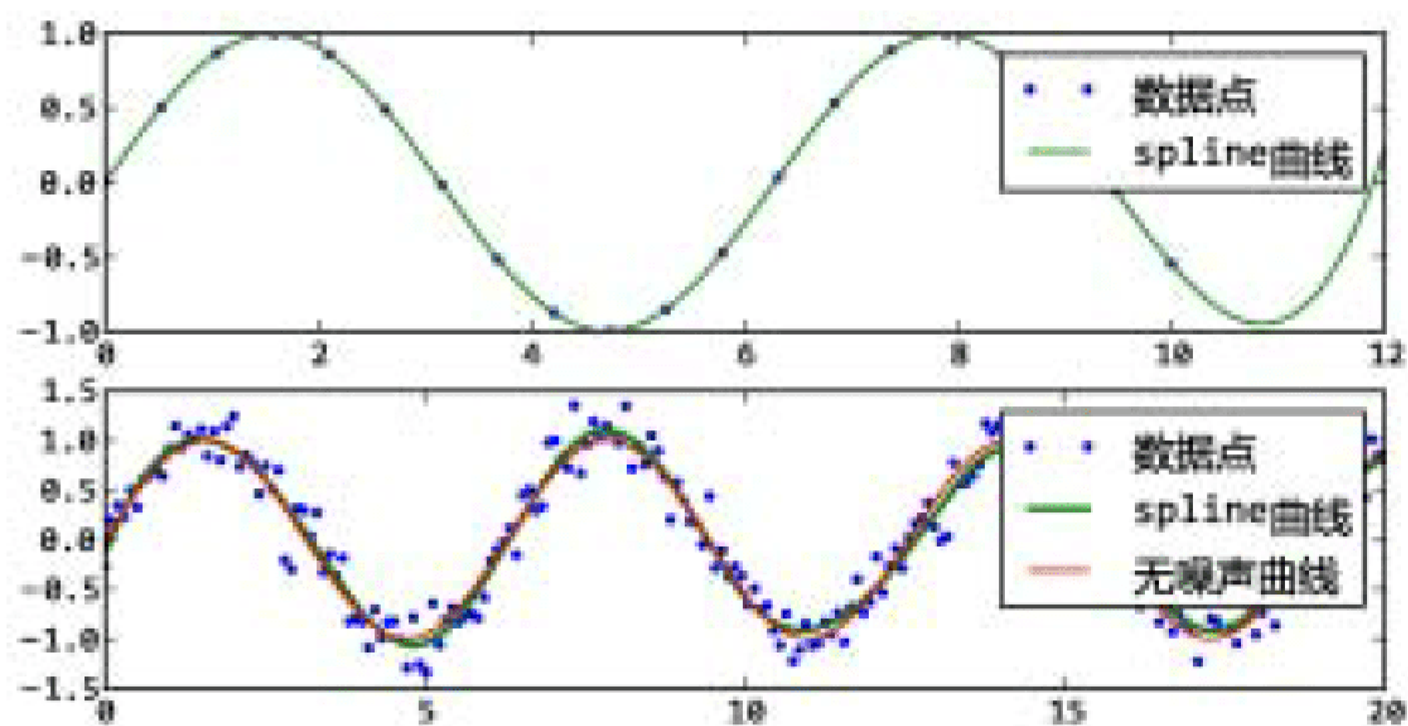


图 3-5　使用 UnivariateSpline 进行插值：外推(上)，数据拟合(下)

3.3.3　二维插值

使用 interp2d()可以进行二维插值运算，它的调用形式如下：

```
interp2d(x, y, z, kind='linear', ...)
```

其中：x、y、z 都是一维数组，如果传入的是多维数组，就先将它转换为一维数组；kind 参数指定插值运算的阶数，可以为'linear'、'cubic'或'quintic'。

下面的例子对某个函数曲面上的网格点进行二维插值，效果如图 3-6 所示(见封二彩插)。其中，左图显示插值之前的数据，右图显示插值运算后得到的结果。

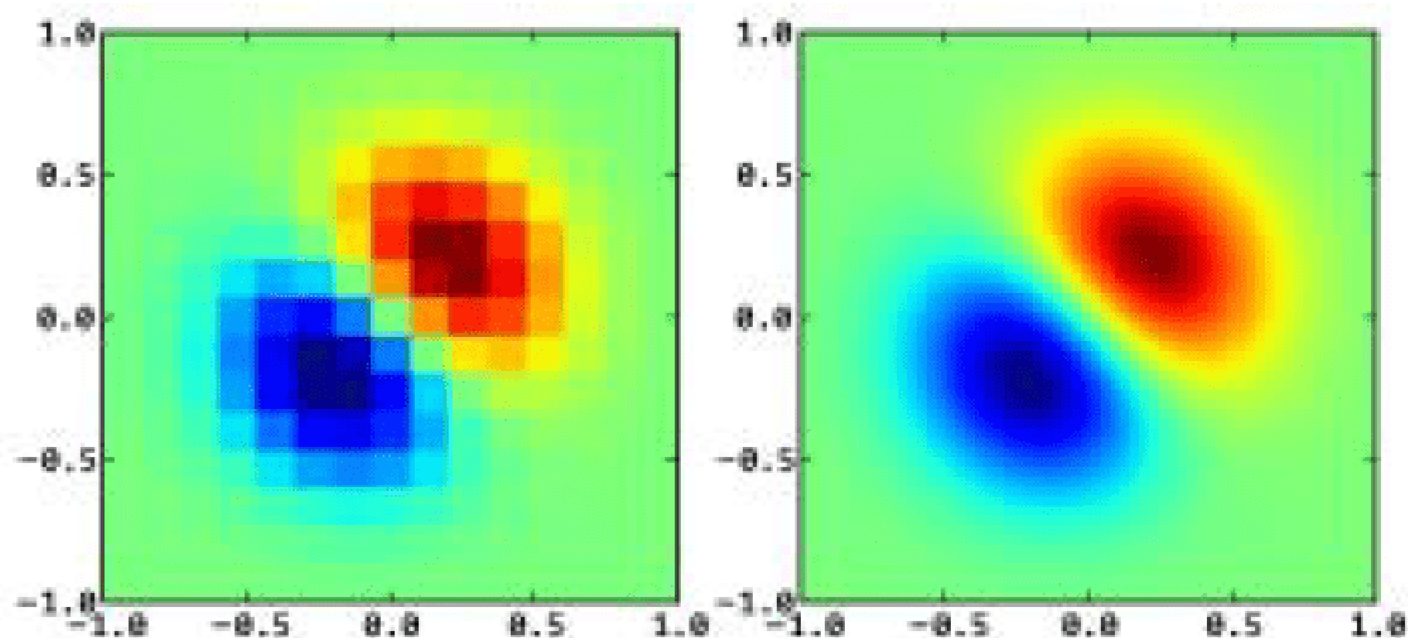


图 3-6　使用 interp2d 进行二维插值



scipy_interp2d.py
使用 interp2d 函数进行二维插值

```
def func(x, y): ❶
    return (x+y)*np.exp(-5.0*(x**2 + y**2))
```

```
# X-Y 轴分为 15*15 的网格
y, x = np.mgrid[-1:1:15j, -1:1:15j] ❷
fvals = func(x,y) # 计算每个网格点上的函数值

# 二维插值
newfunc = interpolate.interp2d(x, y, fvals, kind='cubic') ❸

# 计算 100*100 的网格上的插值
xnew = np.linspace(-1,1,100)
ynew = np.linspace(-1,1,100)
fnew = newfunc(xnew, ynew) ❹
```

❶func 是计算曲面上各点高度的函数。❷计算 X、Y 轴在-1 到 1 范围之内，大小为 15*15 的等间距网格上各点的高度。注意所得到的二维数组 fvals 的第 0 轴与 Y 轴对应，第一轴与 X 轴对应。❸使用网格上各点的 X、Y 和 Z 轴的坐标创建 interp2d 对象，这里我们使用二阶插值曲面。❹interp2d 对象可以像函数一样调用，我们用它计算插值曲面在一个更密的网格中的高度值。注意这里的参数是两个一维数组，分别指定网格的 X-Y 轴坐标，而不需要通过 mgrid 创建网格坐标数组。

interp2d 只能对网格形状的取样值进行插值运算，如果需要对随机散列的取样点进行插值，就需要使用径向基函数(Radial Basis Function，简称 RBF)插值算法。RBF 支持多维散列点的插值运算，下面以二维插值为例演示 RBF 的使用方法。



scipy_rbf.py

使用 RBF 对随机取样点进行二维插值

```
def func(x,y):
    return (x+y)*np.exp(-5.0*(x**2 + y**2))

# 计算曲面函数上 100 个随机分布的点
x = np.random.uniform(-1.0, 1.0, size=100)
y = np.random.uniform(-1.0, 1.0, size=100)
fvals = func(x,y) ❶

# 使用 RBF 进行插值运算
newfunc = interpolate.Rbf(x, y, fvals, function='multiquadric') ❷
ynew, xnew = np.mgrid[-1:1:100j, -1:1:100j] # 插值结果的网格
fnew = newfunc(xnew, ynew) ❸
truevals = func(xnew, ynew) # 函数的真实值
```

❶随机计算曲面上的 100 个点，❷使用随机点创建一个 RBF 对象，并通过 function 参数指定所使用的径向基函数。读者可以查看帮助文档以了解更多的径向基函数选项。❸RBF 对象也可以像函数那样被调用，我们用它计算更密的网格上各点的值。它的两个参数是指定 X-Y 轴坐

标的两个数组。和 interp2d 对象不同的是，它不会自动产生网格上的各点，因此为了使用等距的正交网格，我们使用 mgird 对象创建这两个数组。

程序的运行结果如图 3-7 所示。左图为函数曲面的真实值，右图为径向基函数插值算法通过曲面上的随机点重建的曲面(见封二彩插)。

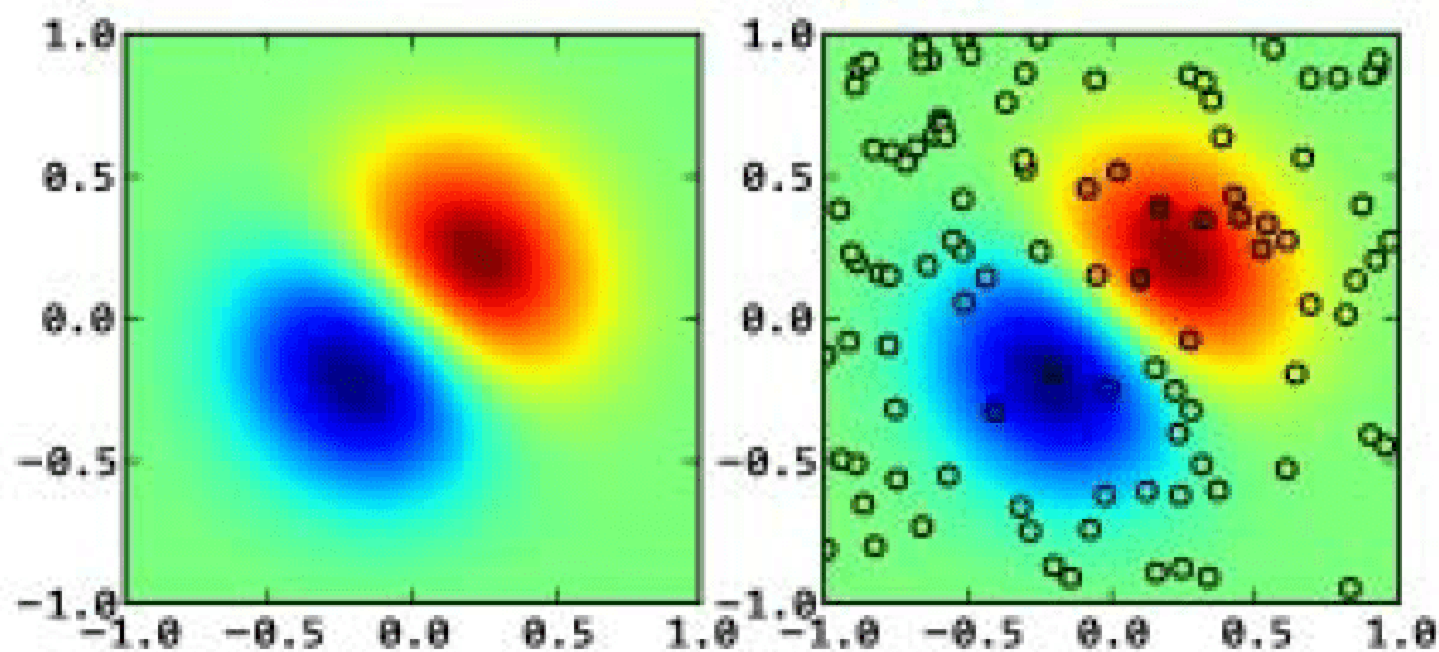


图 3-7　使用径向基函数插值算法对随机取样点进行二维插值

3.4　数值积分——integrate

SciPy 的 integrate 模块提供了几种数值积分算法，其中包括对常微分方程组(ODE)的数值积分。本节以计算球体体积和洛伦茨吸引子轨迹为例介绍 integrate 模块的用法。

3.4.1　球的体积

数值积分是对定积分的数值求解，例如可以利用数值积分计算某个形状的面积。先考虑一下如何计算半径为 1 的半圆的面积。根据圆的面积公式，其面积应该等于 $\pi/2$ 。单位半圆的曲线方程为 $y=\sqrt{1-x^2}$ ，可以通过下面的 half_circle()进行计算：



scipy_integrate.py
用数值积分求圆的面积和球的体积

```
def half_circle(x):
    return (1-x**2)**0.5
```

最简单的数值积分算法就是将要积分的面积分为许多小矩形，然后计算这些矩形的面积之和。下面使用这种方法，将 X 轴上-1 到 1 的区间分为 10000 等份，然后计算面积和：

```
>>> N = 10000
>>> x = np.linspace(-1, 1, N)
```

```
>>> dx = x[1] - x[0]
>>> y = half_circle(x)
>>> 2 * dx * np.sum(y) # 面积的两倍
3.1415893269307378
```

也可以用 NumPy 的 trapz()计算半圆上由各点构成的多边形的面积：

```
>>> np.trapz(y, x) * 2 # 面积的两倍
3.1415893269316042
```

trapz()计算的是以(x,y)为顶点坐标的折线与 X 轴所夹的面积。

如果使用 SciPy 的 integrate 模块中的数值积分函数 quad()，将能得到非常精确的结果：

```
>>> from scipy import integrate
>>> pi_half, err = integrate.quad(half_circle, -1, 1)
>>> pi_half*2
3.1415926535897984
```

计算多重定积分可以通过多次调用 quad()来实现，为了调用方便，integrate 模块提供了 dblquad()以进行二重定积分，以及 tplquad()用于进行三重定积分。下面以计算单位半球体积为例，说明 dblquad()的用法。

单位半球面上的点(x,y,z)满足如下方程：

$$x^2 + y^2 + z^2 = 1$$

因此下面的 half_sphere()可以通过 X-Y 轴坐标计算球面上点的 Z 轴坐标值：

```
def half_sphere(x, y):
    return (1-x**2-y**2)**0.5
```

X-Y 轴平面与此球体的交线为一个单位圆，因此二重积分的计算区间为此单位圆。即对于 X 轴从-1 到 1 进行积分，而对于 Y 轴则从-half_circle(x)到 half_circle(x)进行积分。因此半球体积的二重积分公式为：

$$\int_{-1}^{1} \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \sqrt{1 - x^2 - y^2}\, dydx$$

下面的程序使用 dblquad()计算半球体积：

```
>>> integrate.dblquad(half_sphere, -1, 1,
    lambda x:-half_circle(x),
    lambda x:half_circle(x))
>>> (2.0943951023931988, 2.3252456653390915e-14)
```

```
>>> np.pi*4/3/2 # 通过球体体积公式计算半球体积
2.0943951023931953
```

dblquad()的调用参数为：

```
dblquad(func2d, a, b, gfun, hfun)
```

其中，func2d 是需要进行二重积分的函数，它有两个参数，假设分别为 x 和 y。a 和 b 参数指定被积分函数的第一个变量(即 x)的积分区间，而 gfun 和 hfun 参数指定第二个变量(即 y)的积分区间。gfun 和 hfun 是函数，它们通过变量 x 计算出变量 y 的积分区间，这样可以在 X-Y 平面上的任何区间对 func2d 进行积分。

图 3-8 是半球体积的积分示意图。从此示意图可以看出，X 轴的积分区间为-1.0 到 1.0，对于 X 轴上的某点 x0，通过对 Y 轴的积分可以计算出图中深色的垂直切面的面积，因此 Y 轴的积分区间如图中的点线所示。

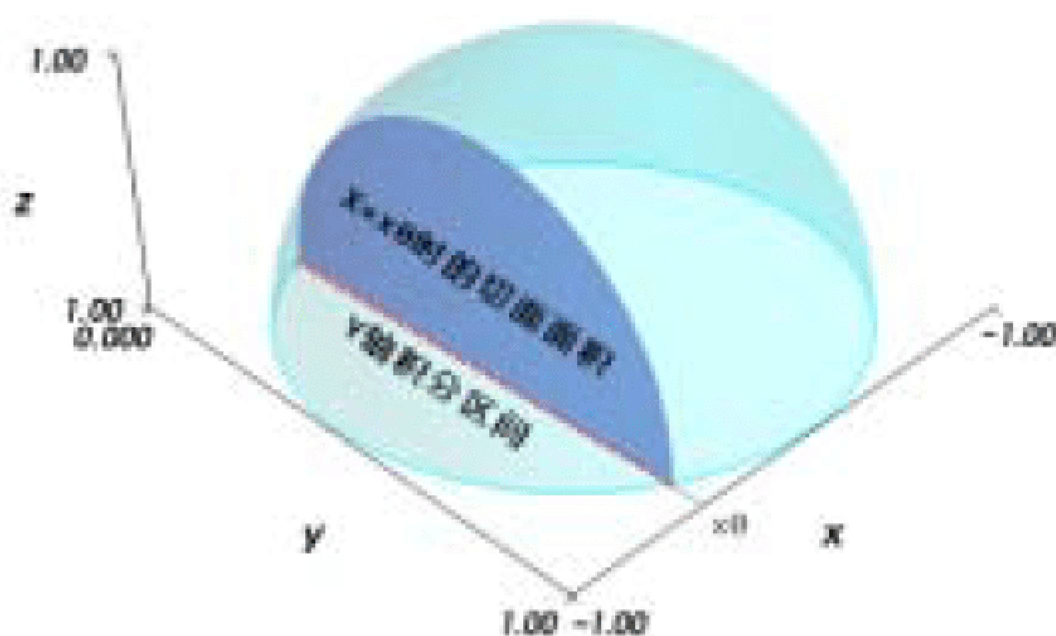


图 3-8　半球体积的双重定积分示意图

3.4.2　解常微分方程组

integrate 模块还提供了对常微分方程组进行积分的函数 odeint()。下面我们看看如何用它计算洛伦茨吸引子的轨迹。洛伦茨吸引子由下面的三个微分方程定义：

$$\frac{dx}{dt}=\sigma\cdot(y-x),\frac{dy}{dt}=x\cdot(\rho-z)-y,\frac{dz}{dt}=xy-\beta z$$



http://bzhang.lamost.org/website/archives/lorenz_attactor
洛伦茨吸引子的详细介绍

这三个方程定义了三维空间中各个坐标点上的速度矢量。从某个坐标开始沿着速度矢量进行积分，就可以计算出无质量点在此空间中的运动轨迹。其中，σ、ρ、β 为三个常数，不同的参数可以计算出不同的运动轨迹：x(t)、y(t)、z(t)。当参数为某些值时，轨迹出现混沌现象。即微小的初值差别也会显著地影响运动轨迹。下面是洛伦茨吸引子的轨迹计算和绘制程序：



Scipy_odeint_lorenz.py
使用 odeint 计算洛伦茨吸引子轨迹

```
from scipy.integrate import odeint
import numpy as np

def lorenz(w, t, p, r, b): ❶
    # 给出位置矢量 w，以及三个参数 p、r、b，计算出 dx/dt、dy/dt、dz/dt 的值
    x, y, z = w.tolist()
    # 直接与 lorenz 的计算公式对应
    return p*(y-x), x*(r-z)-y, x*y-b*z

t = np.arange(0, 30, 0.01) # 创建时间点
# 调用 ode 对 lorenz 进行求解，用两个不同的初始值
track1 = odeint(lorenz, (0.0, 1.00, 0.0), t, args=(10.0, 28.0, 3.0)) ❷
track2 = odeint(lorenz, (0.0, 1.01, 0.0), t, args=(10.0, 28.0, 3.0)) ❸

# 绘图
from mpl_toolkits.mplot3d import Axes3D ❹
import matplotlib.pyplot as plt

fig = plt.figure()
ax = Axes3D(fig)
ax.plot(track1[:,0], track1[:,1], track1[:,2])
ax.plot(track2[:,0], track2[:,1], track2[:,2])
plt.show()
```

❶程序中首先定义一个函数 lorenz()，它的任务是计算出某个坐标点在各个方向上的微分值，可以直接根据洛伦茨吸引子的公式得出。

❷❸使用不同的位移初始值两次调用 odeint()，对微分方程求解。odeint()有许多参数，这里用到的 4 个参数分别为：

- lorenz：它是计算某个位置上各个方向的速度的函数。
- (0.0, 1.0, 0.0)：位置初始值，它是计算常微分方程所需的各个变量的初始值。
- t：表示时间的数组，odeint()对此数组中的每个时间点进行求解，得出所有时间点的位置。
- args：这些参数直接传递给 lorenz()，因此它们在整个积分过程中都是常量。

❹最后通过 matplotlib 的三维绘图模块[①]绘制出 odeint()后得到的轨迹。如图 3-9 所示，即使初始值只相差 0.01，两条运动轨迹也是完全不同的。

① 有关 matplotlib 的三维绘图模块将在 5.4.7 节中进行介绍。

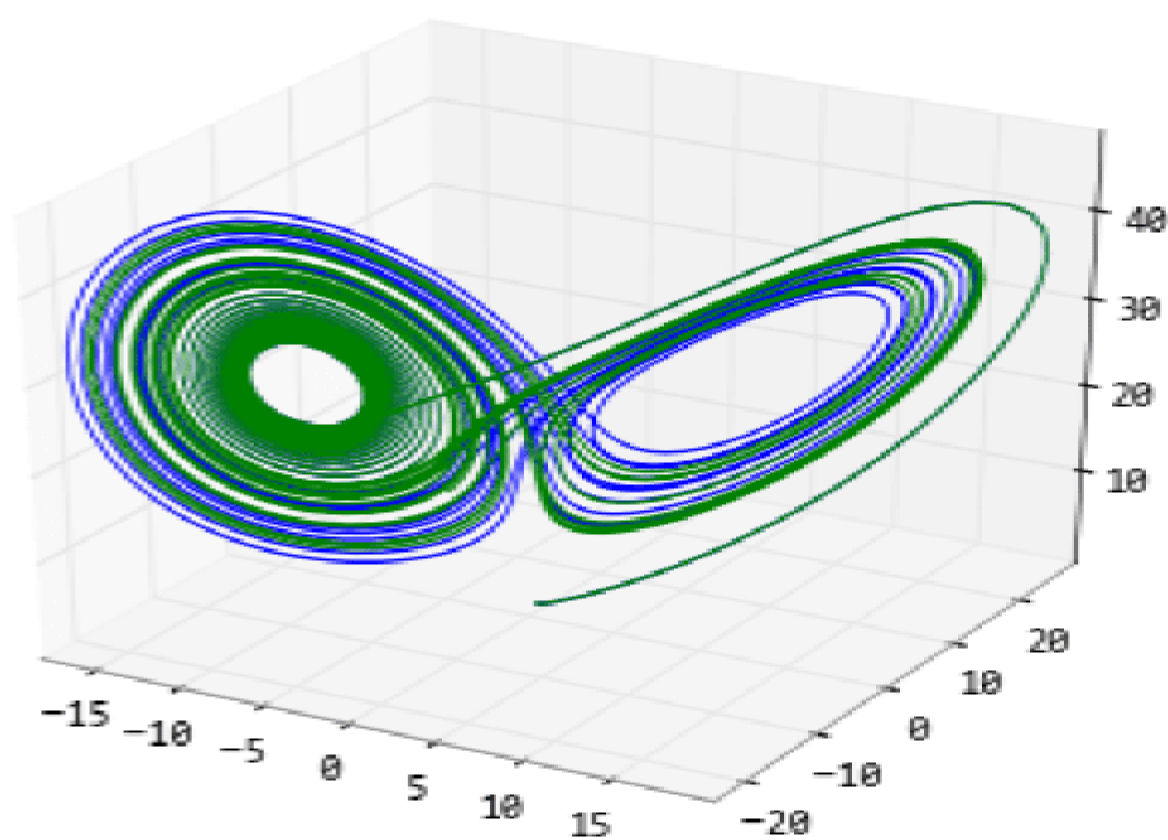


图 3-9　用 odeint()对洛伦茨吸引子微分方程进行数值求解后得到的运动轨迹

3.5　信号处理——signal

SciPy 的 signal 模块提供了许多信号处理方面的函数，包括卷积运算、B 样条、滤波以及滤波器设计等方面的内容。本节简单介绍中值滤波和 IIR 带通滤波器的设计。

3.5.1　中值滤波

中值滤波能够比较有效地消除声音信号中的瞬间噪声或者图像中的斑点噪声。在 signal 模块中，medfilt()对一维信号进行中值滤波，而 medfilt2d()对二维信号进行中值滤波。在 scipy.ndimage 模块中，另有针对多维图像的中值滤波器，这里简单演示 medfilt()中值滤波的效果。



scipy_signal_noise_filter.py
用中值滤波剔除瞬间噪声

首先导入 signal 模块:

```
>>> import scipy.signal as signal
```

然后创建一个带有随机的瞬间噪声的正弦波：

```
>>> t = np.arange(0, 20, 0.1)
>>> x = np.sin(t)
>>> x[np.random.randint(0, len(t), 20)] += np.random.standard_normal(20)*0.6
```

调用 medfilt()进行中值滤波：

```
>>> x2 = signal.medfilt(x, 5)
```

第二个参数为计算中值的窗口大小，它必须是一个奇数。medfilt()将信号中的每个元素都替换为其窗口内的中值。

最后绘制原始信号和滤波信号，为了便于比较，图 3-10 中将滤波之后的信号统一向上偏移了 0.5(见封二彩插)。

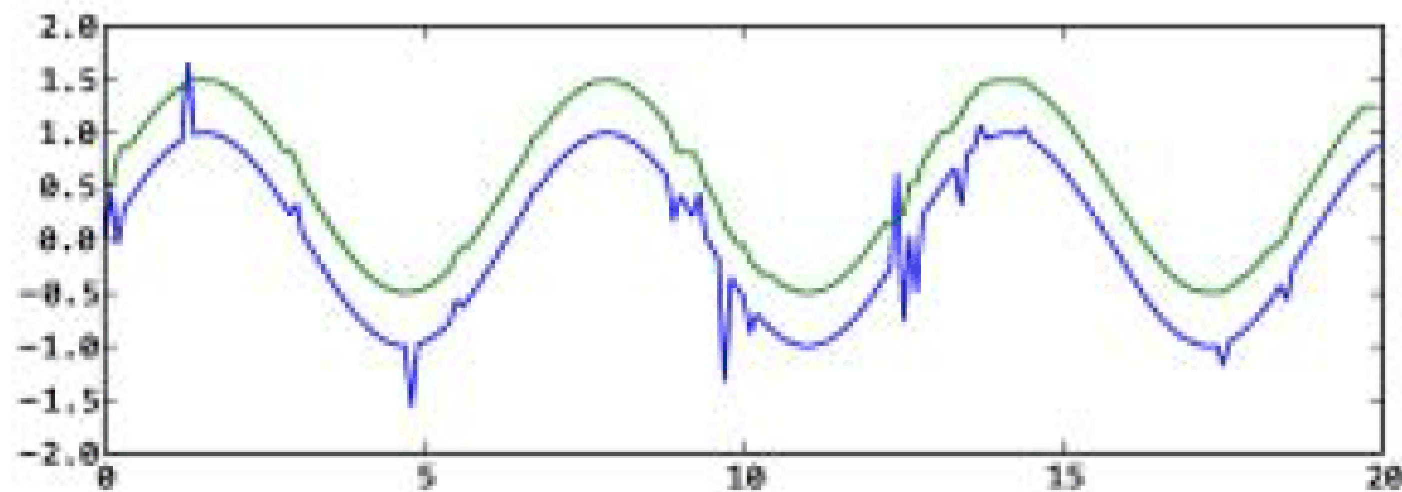


图 3-10　使用中值滤波剔除瞬间噪声

中值滤波是排序滤波的一个特例。使用排序滤波可以将元素替换为其窗口内指定排序顺序的元素。其调用形式如下：

```
order_filter(a, domain, rank)
```

其中：a 是一个多维数组；domain 是维数和 a 相同的数组，它指定窗口的范围；rank 是一个非负整数，用来选择窗口中元素排序后的值，0 表示选择最小值，1 表示选择第二小的值。

中值滤波也可以用 order_filter()来计算，注意 domain 参数是一个长度为 5、值全为 1 的数组：

```
>>> x3 = signal.order_filter(x, np.ones(5), 3)
>>> np.all(x2==x3)
True
```



scipy_signal_order_filter.py
order_filter 用法演示

3.5.2　滤波器设计

signal 模块提供了许多滤波器设计的函数。在下面的实例中，我们设计一个 IIR 带通滤波器，并查看其频率响应，最后使用它对频率扫描信号进行滤波计算。



scipy_signal_bandpass.py
设计带通 IIR 滤波器并用频率扫描波测试其频率响应

首先用 iirdesign()设计一个 IIR 带通滤波器：

```
>>> b, a = signal.iirdesign([0.2, 0.5], [0.1, 0.6], 2, 40)
```

这个滤波器的通带为 0.2f_0 到 0.5 f_0，阻带为小于 0.1 f_0 和大于 0.6 f_0，其中 f_0 为信号取样频率的一半。如果取样频率为 8k Hz，那么这个带通滤波器的通带为 800 Hz 到 2k Hz。通带的最大增益衰减为 2 dB，阻带的最小增益衰减为 40 dB，即通带的增益浮动在 2 dB 之内，阻带至少有 40 dB 的衰减。

iirdesgin()返回两个数组 b 和 a，它们分别是 IIR 滤波器的分子和分母部分的系数。其中 a[0]恒等于 1。下面通过调用 freqz()计算所得到的滤波器的频率响应：

```
>>> w, h = signal.freqz(b, a)
```

freqz()返回两个数组 w 和 h，其中 w 是圆频率数组，通过 $\omega f_0 / \pi$ 可以计算出其对应的实际频率。h 是 w 中对应频率点的响应，它是一个复数数组，其幅值表示滤波器的增益特性，相角表示滤波器的相位特性。

下面计算 h 的增益特性，并使用 dB 进行度量。由于 h 中存在幅值几乎为 0 的值，因此先用 clip()对其裁剪之后，再调用对数函数，避免计算出错。

```
>>> power = 20*np.log10(np.clip(np.abs(h), 1e-8, 1e100))
```

通过下面的语句可以绘制出如图 3-11(上)所示的滤波器增益特性，这里假设取样频率为 8k Hz：

```
>>> import pylab as pl
>>> pl.plot(w/np.pi*4000, power)
```

在实际运用中，为了测量未知系统的频率特性，经常将频率扫描波输入到系统中，观察系统的输出，从而计算其频率特性。下面模拟这一过程。

为了调用 chirp()产生频率扫描波形的数据，首先需要产生一个表示取样时间的等差数组。下面的语句产生 2 秒钟、取样频率为 8k Hz 的取样时间数组：

```
>>> t = np.arange(0, 2, 1/8000.0)
```

然后调用 chirp()得到 2 秒钟的频率扫描波形的数据。频率扫描波的开始频率 f0 为 0 Hz，结束频率 f1 为 4k Hz，到达 4k Hz 的时间为 2 秒，使用数组 t 作为取样时间点。

```
>>> sweep = signal.chirp(t, f0=0, t1 = 2, f1=4000.0)
```

最后调用 lfilter()计算频率扫描波形经过带通滤波器之后的结果：

```
>>> out = signal.lfilter(b, a, sweep)
```

为了和系统的增益特性图进行比较，需要获取输出波形的包络，因此下面先将输出波形数据转换为能量值：

```
>>> out = 20*np.log10(np.abs(out))
```

为了计算包络，找到所有能量大于前后两个取样点(局部最大点)的下标：

```
>>> index = np.where(np.logical_and(out[1:-1] > out[:-2], out[1:-1] > out[2:]))[0] + 1
```

最后将时间转换为对应的频率，绘制所有局部最大点的能量值：

```
>>> pl.plot(t[index]/2.0*4000, out[index] )
```

图 3-11 显示了 freqz()计算的频谱和频率扫描波得到的频率特性，可以看出结果是一致的。

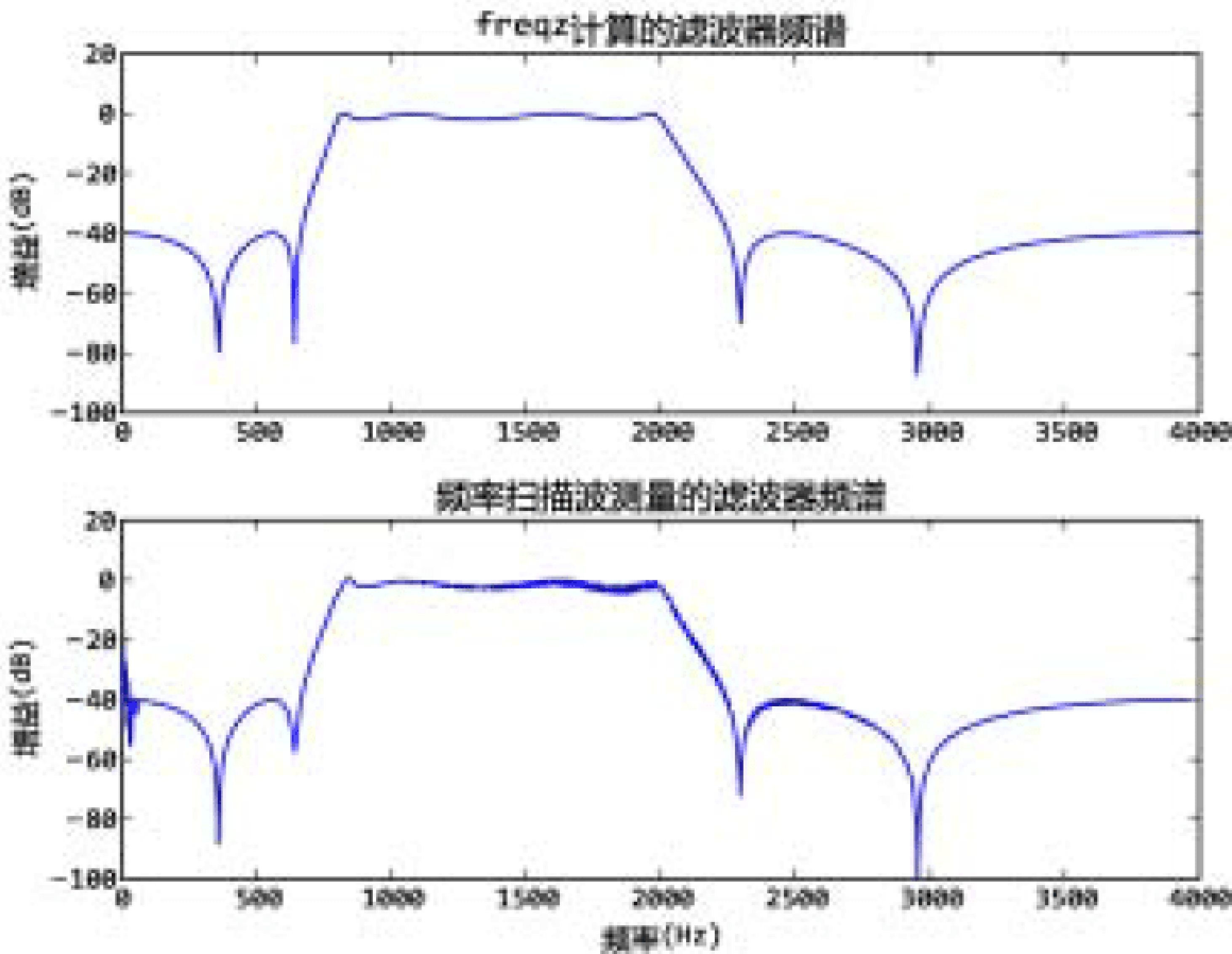


图 3-11　带通 IIR 滤波器的频率响应和频率扫描波计算的结果比较

3.6　图像处理——ndimage

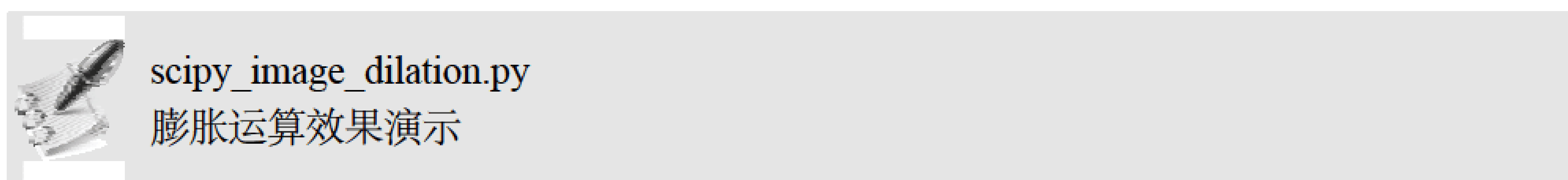
scipy.ndimage 是一个处理多维图像的函数库，其中又包括以下几个模块：

- filters：图像滤波器
- fourier：傅里叶变换
- interpolation：图像的插值、旋转及仿射变换等
- measurements：图像相关信息的测量
- morphology：形态学图像处理

本节介绍如何使用 morphology 模块实现二值图像处理。二值图像中每个像素的颜色只有两种：黑色和白色。在 NumPy 中可以用二维布尔数组表示：False 表示黑色，True 表示白色。也可以用无符号单字节整型(uint8)数组表示：0 表示黑色，255 表示白色。

3.6.1 膨胀和腐蚀

二值图像最基本的形态学运算是膨胀和腐蚀。膨胀运算是将与某物体(白色区域)接触的所有背景像素(黑色区域)合并到该物体中的过程。简单地说，就是对原始图像中的每个白色像素进行处理，将其周围的黑色像素都设置为白色像素。这里的“周围”是一个模糊概念，在实际运算时，需要明确给出“周围”的定义。图 3-12 是膨胀运算的一个例子，其中左图是原始图像，中间的图是四连通定义的“周围”的膨胀效果，右图是八连通定义的“周围”的膨胀效果。图中用灰色方块表示由膨胀处理添加进物体的像素。



scipy_image_dilation.py
膨胀运算效果演示

图 3-12 原始图像(左)、四连通膨胀(中)、八连通膨胀(右)

四连通包括上下左右 4 个像素，而八连通则还包括 4 个斜线方向上的邻接像素。它们都可以使用下面的正方形矩阵来定义，其中正中心的元素表示当前要进行运算的像素，而其周围的 1 和 0 表示对应位置的像素是否算其“周围”像素。这种矩阵描述了周围像素和当前像素之间的关系，被称为结构元素(structuring element)。

```
四连通      八连通
0 1 0      1 1 1
1 1 1      1 1 1
0 1 0      1 1 1
```

假设数组 a 是一个表示二值图像的数组，可以用如下语句对其进行膨胀运算：

```
>>> import scipy.ndimage.morphology as m
>>> b = m.binary_dilation(a)
```

binary_dilation()默认使用四连通进行膨胀运算，通过 structure 参数可以指定其他的结构元素。下面是进行八连通膨胀运算的语句：

```
>>> c = m.binary_dilation(a, structure=[[1,1,1],[1,1,1],[1,1,1]])
```

通过设置不同的结构元素，能够制作出各种不同的效果。图 3-13 显示了三种不同结构元素

的膨胀效果，图中的结构元素分别为：

```
 左      中     右
0 0 0  0 1 0  0 1 0
1 1 1  0 1 0  0 1 0
0 0 0  0 1 0  0 0 0
```

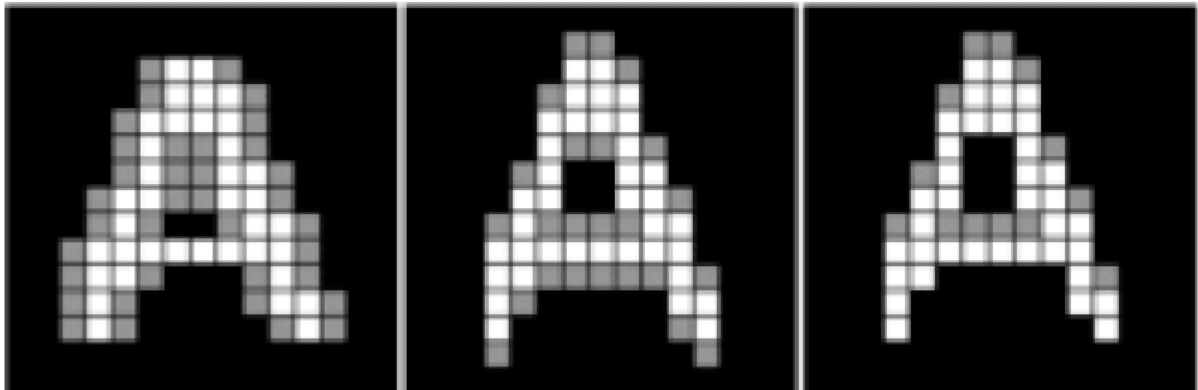


图 3-13　不同的结构元素产生不同的膨胀效果

binary_erosion()的腐蚀运算正好和膨胀运算相反，它将“周围”有黑色像素的白色像素设置为黑色。图 3-14 是四连通和八连通腐蚀的效果，图中用灰色方块表示被腐蚀的像素。

scipy_image_erosion.py
腐蚀运算效果演示

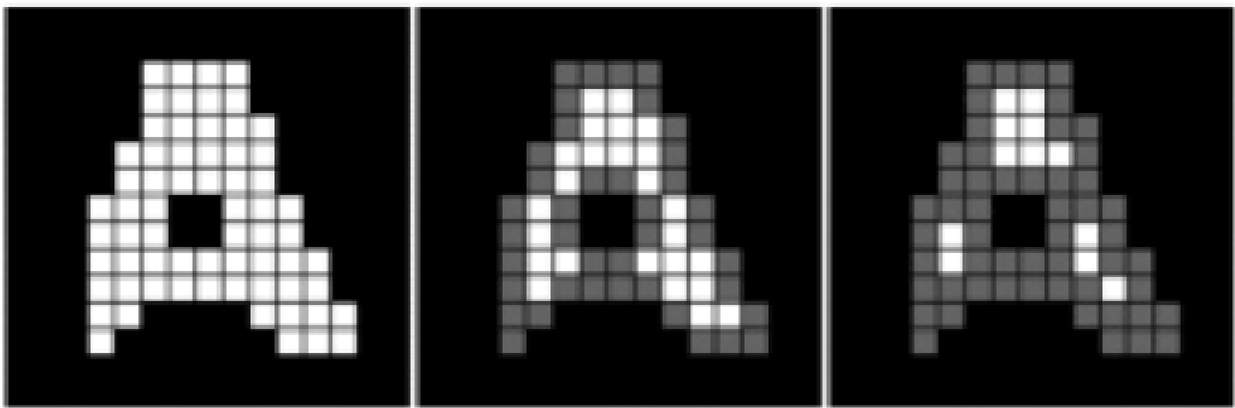


图 3-14　原始图像(左)、四连通腐蚀(中)、八连通腐蚀(右)

3.6.2　Hit 和 Miss

Hit 和 Miss 是二值形态学图像处理中最基本的运算，因为几乎所有其他的运算都可以用 Hit 和 Miss 的组合推演出来。它对图像中每个像素周围的像素进行模式判断，如果周围像素的黑白模式符合指定的模式，就将此像素设为白色，否则设为黑色。因为它需要同时对白色和黑色像素进行判断，因此需要指定两个结构元素。进行 Hit 和 Miss 运算的 binary_hit_or_miss()的调用形式如下：

```
binary_hit_or_miss(input, structure1=None, structure2=None, ...)
```

其中，structure1 参数指定白色像素的结构元素，structure2 参数则指定黑色像素的结构元

素。图 3-15 是 binary_hit_or_miss()的运算结果。其中，左图为原始图像，中图为使用下面两个结构元素进行运算的结果：

```
白色结构元素      黑色结构元素
  0 0 0            1 0 0
  0 1 0            0 0 0
  1 1 1            0 0 0
```

在这两个结构元素中，0 表示不关心其对应位置的像素的颜色，1 表示其对应位置的像素必须为结构元素所表示的颜色。因此，通过这两个结构元素可以找到“下方三个像素为白色，并且左上方像素为黑色的白色像素”。

与图 3-15(右)对应的结构元素如下，通过它可以找到“下方三个像素为白色、左上方像素为黑色的黑色像素”：

```
白色结构元素      黑色结构元素
  0 0 0            1 0 0
  0 0 0            0 1 0
  1 1 1            0 0 0
```

scipy_image_hitmiss.py
Hit 和 Miss 运算的效果演示

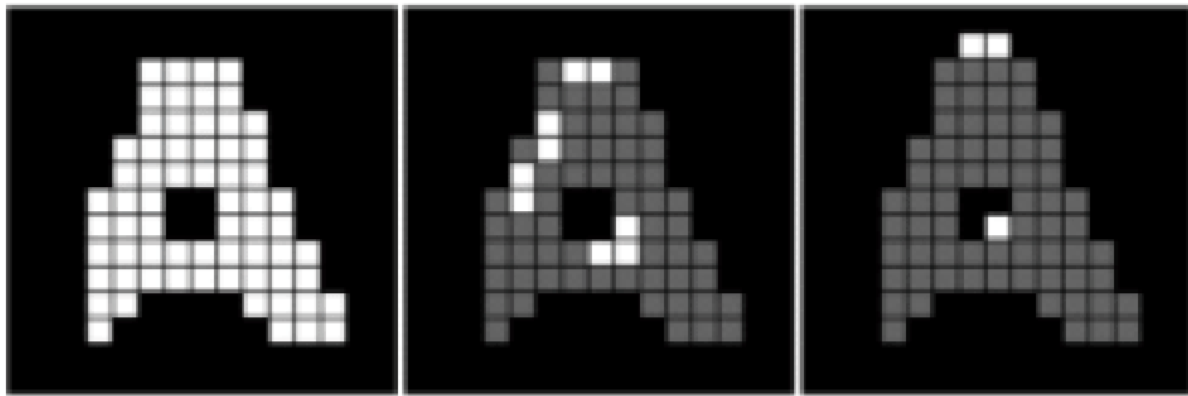


图 3-15　Hit 和 Miss 运算的效果演示

使用 Hit 和 Miss 运算的组合，可以实现很复杂的图像处理。例如文字识别中常用的细线化运算，就可以用一系列的 Hit 和 Miss 运算来实现。图 3-16 显示了细线化处理的效果。

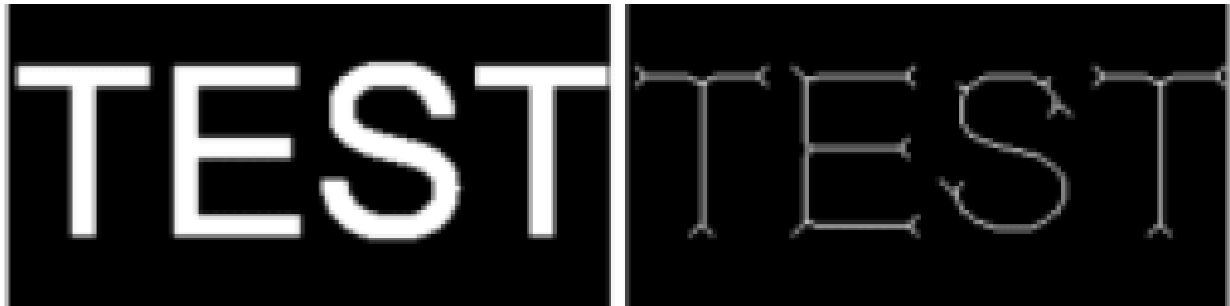


图 3-16　使用 Hit 和 Miss 运算实现细线化处理

细线化算法的实现程序如下，这里只列出其中真正进行细线化算法的函数 skeletonize()：



scipy_image_skeletonization.py
使用 Hit 和 Miss 运算实现细线化处理效果

```
def skeletonize(img):
    h1 = np.array([[0, 0, 0],[0, 1, 0],[1, 1, 1]]) ❶
    m1 = np.array([[1, 1, 1],[0, 0, 0],[0, 0, 0]])
    h2 = np.array([[0, 0, 0],[1, 1, 0],[0, 1, 0]])
    m2 = np.array([[0, 1, 1],[0, 0, 1],[0, 0, 0]])
    hit_list = []
    miss_list = []
    for k in range(4): ❷
        hit_list.append(np.rot90(h1, k))
        hit_list.append(np.rot90(h2, k))
        miss_list.append(np.rot90(m1, k))
        miss_list.append(np.rot90(m2, k))
    img = img.copy()
    while True:
        last = img
        for hit, miss in zip(hit_list, miss_list):
            hm = m.binary_hit_or_miss(img, hit, miss) ❸
            # 从图像中删除 hit_or_miss 所得到的白色像素
            img = np.logical_and(img, np.logical_not(hm)) ❹
        # 如果处理之后的图像和处理前的图像相同，就结束处理
        if np.all(img == last): ❺
            break
    return img
```

❶以图 3-17 所示的两个结构元素为基础，构造 4 个 3*3 的二维数组：h1、m1、h2 和 m2。其中：h1 和 m1 对应图中左边的结构元素，h2 和 m2 对应图中右边的结构元素，h1 和 h2 对应白色结构元素，m1 和 m2 对应黑色结构元素。❷将这些结构元素进行 90°、180°、270° 旋转之后，一共得到 8 个结构元素。

❸依次使用这些结构元素进行 Hit 和 Miss 运算，❹并从图像中删除运算所得到的白色像素，其效果就是依次从 8 个方向删除图像边缘上的像素。❺重复运算直到没有像素可删除为止。

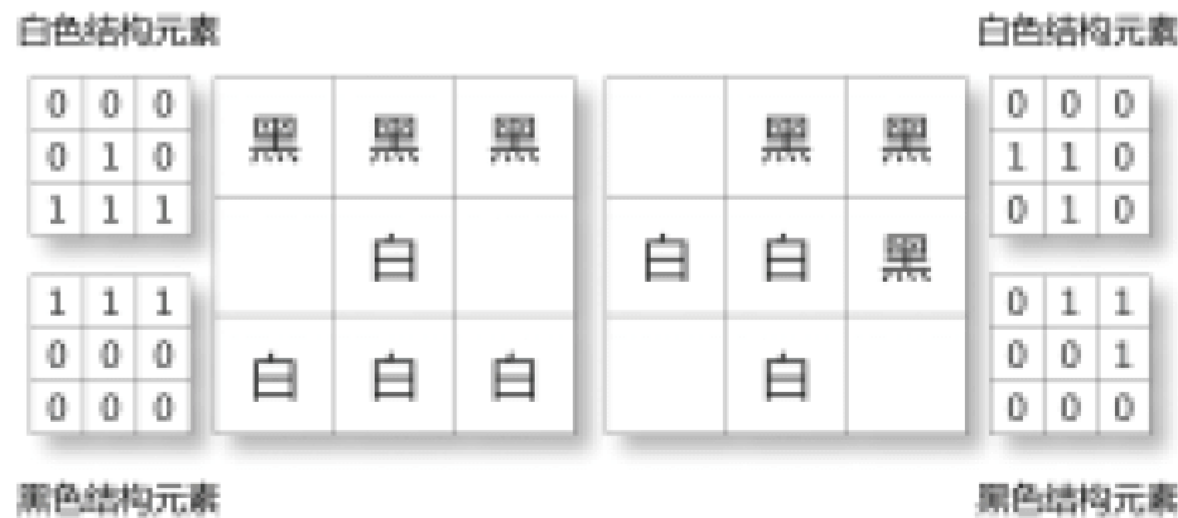


图 3-17　细线化算法的 4 个结构元素

3.7 统计——stats

SciPy 的 stats 模块包含了多种概率分布的随机变量[②]，随机变量分为连续的和离散的两种。所有的连续随机变量都是 rv_continuous 的派生类的对象，而所有的离散随机变量都是 rv_discrete 的派生类的对象。

3.7.1 连续和离散概率分布

可以使用下面的语句获得 stats 模块中所有的连续随机变量：

```
>>> from scipy import stats
>>> [k for k,v in stats.__dict__.items() if isinstance(v, stats.rv_continuous)]
['genhalflogistic','triang','rayleigh','betaprime', ...]
```

连续随机变量对象都有如下方法：

- rvs：对随机变量进行随机取值，可以通过 size 参数指定输出的数组大小。
- pdf：随机变量的概率密度函数。
- cdf：随机变量的累积分布函数，它是概率密度函数的积分。
- sf：随机变量的生存函数，它的值是 1 - cdf(t)。
- ppf：累积分布函数的反函数。
- stat：计算随机变量的期望值和方差。
- fit：对一组随机取样进行拟合，找出最适合取样数据的概率密度函数的系数。



scipy_stats.py
概率密度函数、直方图统计和累积分布函数

下面以正态分布为例，简单介绍随机变量的用法。下面的语句可获得默认正态分布的随机变量的期望值和方差，可以看到，默认情况下它是一个均值为 0、方差为 1 的随机变量：

```
>>> stats.norm.stats()
(array(0.0), array(1.0))
```

norm 可以像函数那样来调用，通过 loc 和 scale 参数可以指定随机变量的偏移和缩放参数。对于正态分布的随机变量来说，这两个参数相当于指定其期望值和标准差[③]：

```
>>> X = stats.norm(loc=1.0, scale=2.0)
>>> x.stats()
(array(1.0), array(4.0))
```

② 这里的随机变量是概率论中的概念，不是 Python 中的变量。
③ 标准差是方差的算术平方根，因此标准差为 2.0 时，方差为 4.0。

下面调用随机变量 X 的 rvs()方法，得到包含一万次随机取样值的数组 x；然后调用 NumPy 的 mean()和 var()，计算此数组的均值和方差，其结果符合随机变量 X 的特性：

```
>>> x = X.rvs(size=10000) # 对随机变量取 10000 个值
>>> np.mean(x) # 期望值
1.0181259658732724
>>> np.var(x)  # 方差
4.00188661640059
```

也可以使用 fit()方法对随机取样序列 x 进行拟合，返回的是与随机取样值最吻合的随机变量的参数：

```
>>> stats.norm.fit(x) # 得到随机序列的期望值和标准差
array([ 1.01810091,  2.00046946])
```

接下来比较随机变量 X 的概率密度函数和对数组 x 进行直方图统计的结果：

```
>>> t = np.arange(-10, 10, 0.01)
>>> pl.plot(t, X.pdf(t)) # 绘制概率密度函数的理论值
>>> p, t2 = np.histogram(x, bins=100, normed=True)
>>> t2 = (t2[:-1] + t2[1:])/2
>>> pl.plot(t2, p) # 绘制统计所得到的概率密度
```

其中，histogram()对数组 x 进行直方图统计，它将数组 x 的取值范围分为 100 个区间，并统计 x 中每个值落入各个区间中的次数。histogram()返回两个数组 p 和 t2，其中 p 表示各个区间取样值出现的频数，由于 normed 参数为 True，因此 p 的值是正规化之后的结果。t2 表示区间，由于其中包括区间起点和终点，因此 t2 的长度为 101。图 3-18(左)显示了概率密度函数和直方图统计的结果，可以看出二者是一致的。

下面的程序绘制随机变量 X 的累积分布函数和数组 p 的累加结果，其结果如图 3-18(右)所示。

```
>>> pl.plot(t, X.cdf(t))
>>> pl.plot(t2, np.add.accumulate(p)*(t2[1]-t2[0]))
```

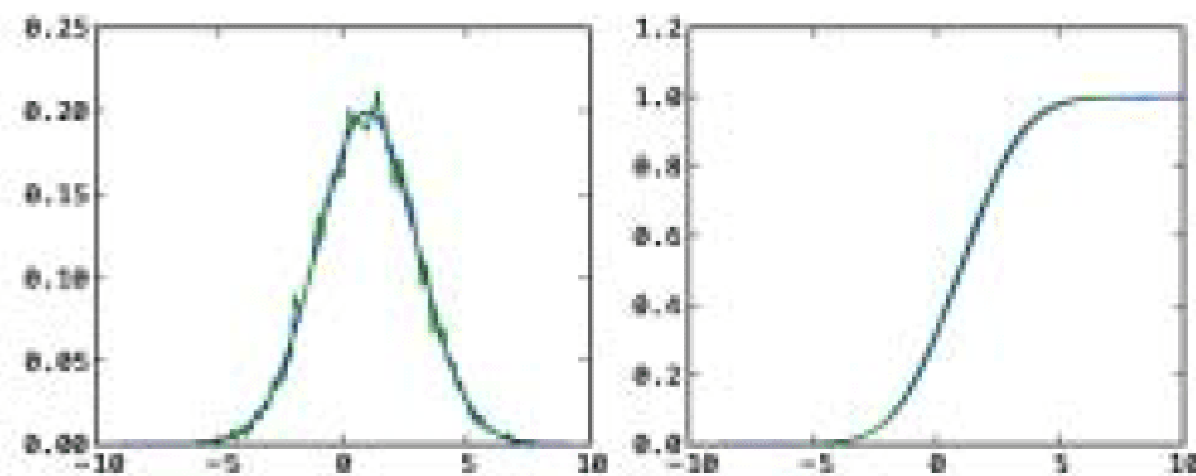


图 3-18　正规分布的概率密度函数(左)和累积分布函数(右)

有些随机分布除了 loc 和 scale 参数之外，还需要额外的形状参数。例如伽玛分布可用于描述等待 k 个独立的随机事件发生所需的时间，k 就是伽玛分布的形状参数。下面计算形状参数 k 为 1 和 2 时伽玛分布的期望值和方差：

```
>>> stats.gamma.stats(1.0)
(array(1.0), array(1.0))
>>> stats.gamma.stats(2.0)
(array(2.0), array(2.0))
```

伽玛分布的尺度参数 θ 和随机事件发生的频率相关，由 scale 参数指定：

```
>>> stats.gamma.stats(2.0, scale=2)
(array(4.0), array(8.0))
```

根据伽玛分布的数学定义可知其期望值为 $k\theta$，方差为 $k\theta^2$。上面的程序验证了这两个公式。

当随机分布有额外的形状参数时，它所对应的 rvs()、pdf()等方法都会增加额外的参数以接收形状参数。例如下面的程序调用 rvs()对 $k=2, \theta=2$ 的伽玛分布取 4 个随机值：

```
>>> x = stats.gamma.rvs(2, scale=2, size=4)
>>> x
array([ 2.20814048,  3.56652153,  4.30088176,  0.68262888])
```

接下来调用 pdf()，查看上面 4 个随机值所对应的概率密度：

```
>>> stats.gamma.pdf(x, 2, scale=2)
array([ 0.18301012,  0.1498734 ,  0.12519094,  0.12130919])
```

也可以先创建将形状参数和尺度参数固定的随机变量，然后再调用其 pdf()计算概率密度：

```
>>> X = stats.gamma(2, scale=2)
>>> X.pdf(x)
array([ 0.18301012,  0.1498734 ,  0.12519094,  0.12130919])
```

当分布函数的值域为离散时，我们称之为离散概率分布。例如投掷有 6 个面的骰子时，只能获得 1 到 6 的整数，因此得到的概率分布为离散的。对于离散随机分布，通常使用概率质量函数(PMF)描述其分布情况。

在 stats 库中所有描述离散分布的随机变量都从 rv_discrete 类继承。也可以直接用 rv_discrete 类自定义离散概率分布。例如假设有一个不均匀的骰子，各点出现的概率不相等。可以用下面的数组 x 保存骰子的所有可能值，数组 p 保存每个值出现的概率：

```
>>> x = range(1,7)
>>> p = (0.4, 0.2, 0.1, 0.1, 0.1, 0.1)
```

于是，可以用下面的语句定义表示这个特殊骰子的随机变量，并调用其 rvs()方法投掷此骰子 20 次，获得符合概率 p 的随机数：

```
>>> dice = stats.rv_discrete(values=(x,p))
>>> dice.rvs(size=20)
array([2, 5, 1, 2, 1, 1, 2, 4, 1, 3, 1, 1, 4, 3, 1, 1, 1, 2, 6, 4])
```

3.7.2 二项、泊松、伽玛分布

本节用几个实例程序对概率论中的二项分布、泊松分布以及伽玛分布进行一些实验和讨论。

二项分布是最重要的离散概率分布之一。假设有一种只有两个结果的试验，其成功概率为 p，那么二项分布描述了进行 n 次这样的独立试验而成功 k 次的概率。二项分布的概率质量函数公式如下：

$$f(k;n,p)=\frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$$

例如，可以通过二项分布的概率质量公式计算投掷 5 次骰子出现 3 次 6 点的概率。投掷一次骰子，点数为 6 的概率(即试验成功的概率)为 $p=1/6$，试验次数为 $n=5$。使用二项分布的概率质量函数 pmf()可以很容易计算出现 k 次 6 点的概率。和概率密度函数 pdf()类似，pmf()的第一个参数为随机变量的取值，后面的参数为描述随机分布所需的参数。对于二项分布来说，参数分别为 n 和 p，而取值范围则为 0 到 n 之间的整数。下面的程序计算 k 为 0 到 6 所对应的概率：

```
>>> stats.binom.pmf(range(6), 5, 1/6.0)
array([0.401878, 0.401878, 0.160751, 0.032150, 0.003215, 0.000129])
```

由结果可知：出现 0 或 1 次 6 点的概率为 40.2%，而出现 3 次 6 点的概率为 3.215%。

在二项分布中，如果试验次数 n 很大，而每次试验成功的概率 p 很小，其乘积 np 比较适中，那么试验成功次数的概率可以用泊松分布近似描述。

在泊松分布中，使用 λ 描述单位时间(或单位面积)内随机事件的平均发生率。如果将二项分布中的试验次数 n 看作单位时间内所做的试验次数，那么它和事件出现概率 p 的乘积就是事件的平均发生率，即 $\lambda=np$。泊松分布的概率质量函数公式如下：

$$f(k;\lambda)=\frac{e^{-\lambda}\lambda^k}{k!}$$

下面的程序分别计算二项分布和泊松分布的概率质量函数，结果如图 3-19 所示，可以看出当 n 足够大时，二者是十分接近的(见封二彩插)。



scipy_binom_poisson.py
比较二项分布和泊松分布的概率质量函数

程序中事件平均发生率 λ 恒等于 10。根据二项分布的试验次数 n，计算每次事件出现的概率 $p=\lambda/n$。程序中的运算部分大致如下：

```
>>> _lambda = 10.0
>>> k = np.arange(20)
>>> possion = stats.poisson.pmf(k, _lambda)  # 泊松分布
>>> binom100 = stats.binom.pmf(k, 100, _lambda/100)    # 二项分布 n=100
>>> binom1000 = stats.binom.pmf(k, 1000, _lambda/1000) # 二项分布 n=1000
>>> np.max(np.abs(binom100-possion)) # 计算最大误差
0.006755311103353312
>>> np.max(np.abs(binom1000-possion)) # n 为 1000 时，误差较小
0.00063017540509099912
```

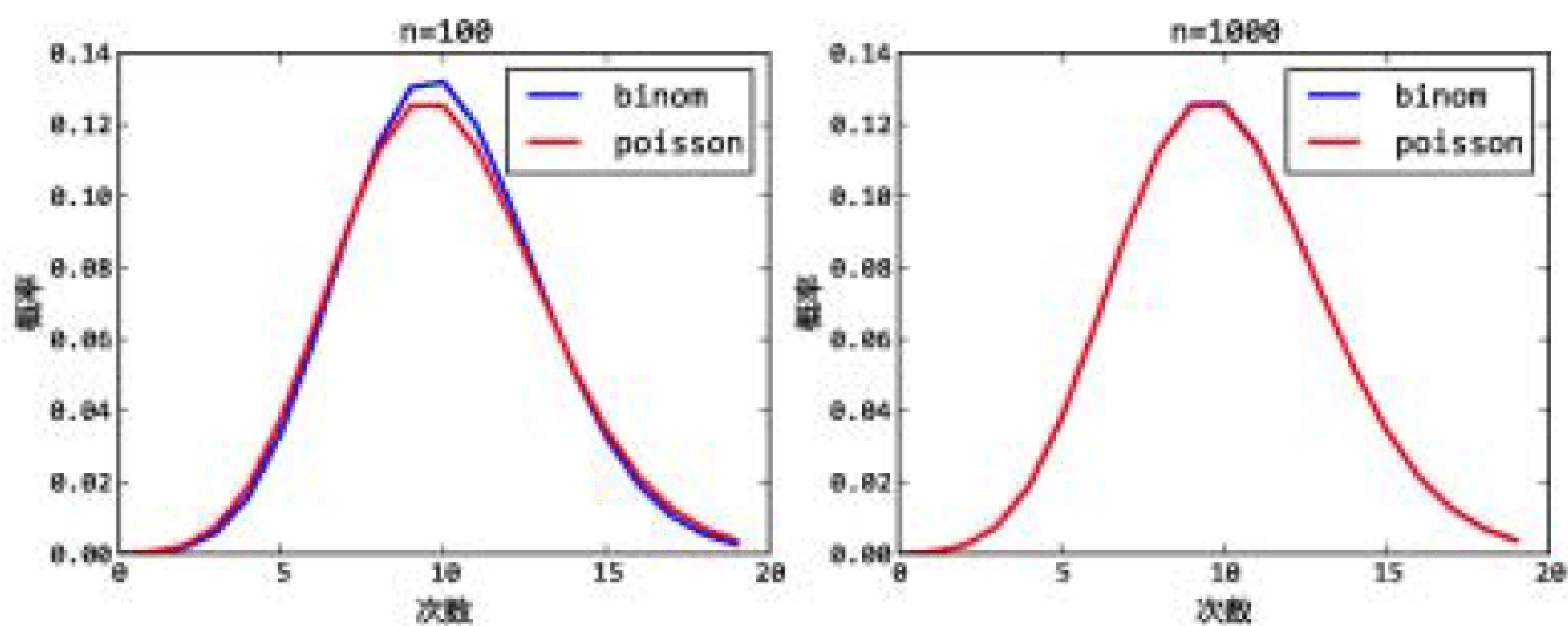


图 3-19　当 n 足够大时二项分布和泊松分布近似相等

泊松分布适合描述单位时间内随机事件发生次数的分布情况。例如某设施在一定时间内的使用次数，机器出现故障的次数，自然灾害发生的次数等等。

为了加深读者对泊松分布概念的理解，下面我们使用随机数模拟泊松分布，并与其概率质量函数进行比较，结果如图 3-20 所示。图 3-20 中，事件每秒的平均发生次数为 10，即 $\lambda=10$。其中左图的观察时间为 1 000 秒，而右图的观察时间为 50 000 秒(见文前彩插)。可以看出：观察时间越长，事件每秒发生的次数就越符合泊松分布。



scipy_poisson_sim.py
模拟泊松分布

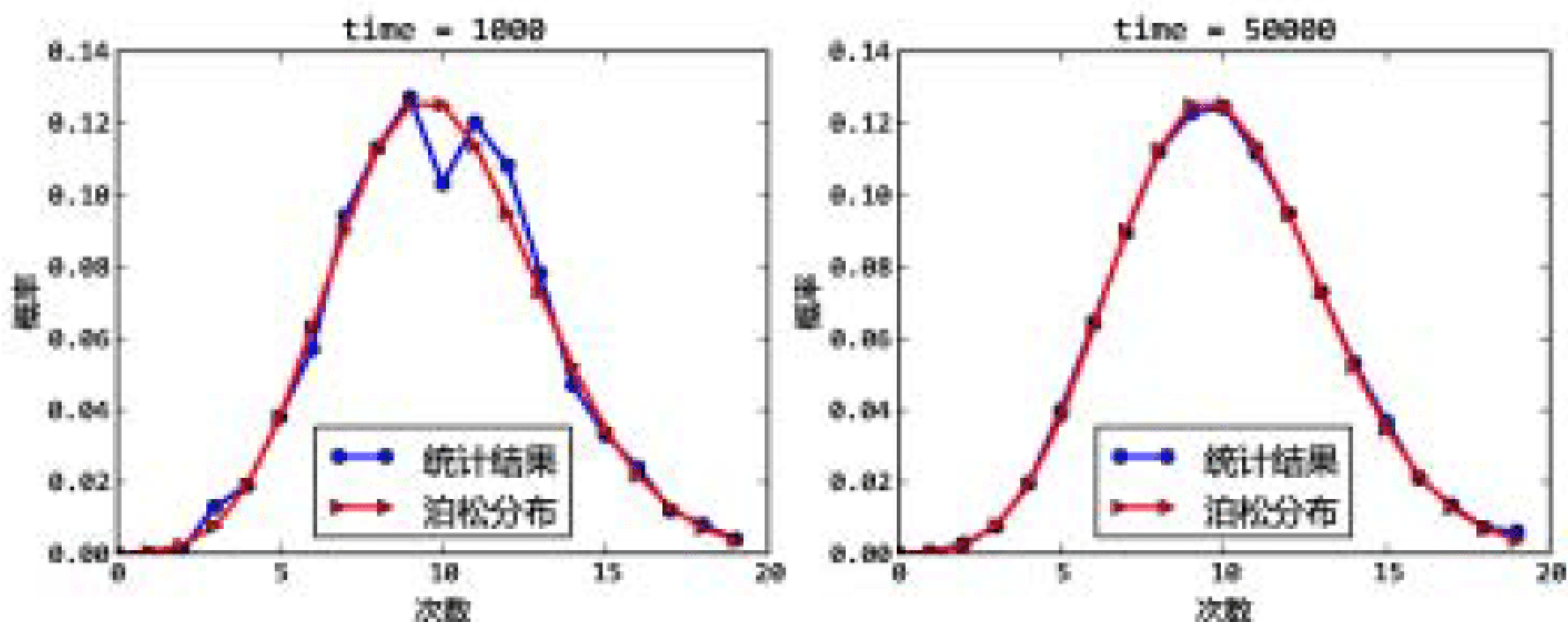


图 3-20　模拟泊松分布

由于上面的程序中包含了许多绘图方面的代码，下面我们直接在 IPython 中介绍泊松分布的模拟过程。首先定义事件发生率 λ 和观察时间：

```
>>> _lambda = 10
>>> time = 10000
```

可以用 NumPy 的随机数生成函数 rand()，产生平均分布于 0 到 time 之间的_lambda*time 个事件所发生的时刻。由于 rand()产生的是 0 到 1 之间的平均分布的随机数，因此需要对其结果扩大 time 倍：

```
>>> t = np.random.rand(_lambda*time)*time
```

用 histogram()可以统计数组 t 中每秒之内事件发生的次数 count：

```
>>> count, time_edges = np.histogram(t, bins=time, range=(0,time))
>>> count
array([10,  9,  8, ..., 11, 10, 18])
```

根据泊松分布的定义，count 数组中数值的分布情况应该符合泊松分布。下面统计事件次数在 0 到 20 区间内的概率分布。当 histogram()的 normed 参数为 True 并且每个统计区间的长度为 1 时，其结果和概率质量函数相等。

```
>>> dist, count_edges = np.histogram(count, bins=20, range=(0,20), normed=True)
>>> poisson = stats.poisson.pmf(x, _lambda)
>>> np.max(np.abs(dist-possion)) # 最大误差很小，符合泊松分布
0.0088356241037075706
```

还可以换一个角度看随机事件的分布问题。我们可以观察相邻两个事件之间时间间隔的分布情况，或者隔 k 个事件的时间间隔的分布情况。根据概率论，事件之间的时间间隔应符合伽玛分布，由于时间间隔可以是任意数值，因此伽玛分布是一种连续概率分布。伽玛分布的概率

密度函数公式如下，它描述第 k 个事件发生所需的等待时间的概率分布。$\Gamma(k)$ 是伽玛函数，当 k 为整数时，它的值和 k 的阶乘 $k!$ 相等。

$$f(X;k,\lambda)=\frac{X^{(k-1)}\lambda^k e^{(-\lambda X)}}{\Gamma(k)}$$

下面的程序模拟了事件的时间间隔的伽玛分布，结果如图 3-21 所示。图 3-21 中的观察时间为 1 000 秒，平均每秒产生 10 个事件。左图中“k=1”，它表示相邻两个事件之间的时间间隔的分布，而“k=2”则表示相隔一个事件的两个事件之间的时间间隔的分布，可以看出它们都符合伽玛分布(见文前彩插)。

scipy_gamma_sim.py
模拟伽玛分布

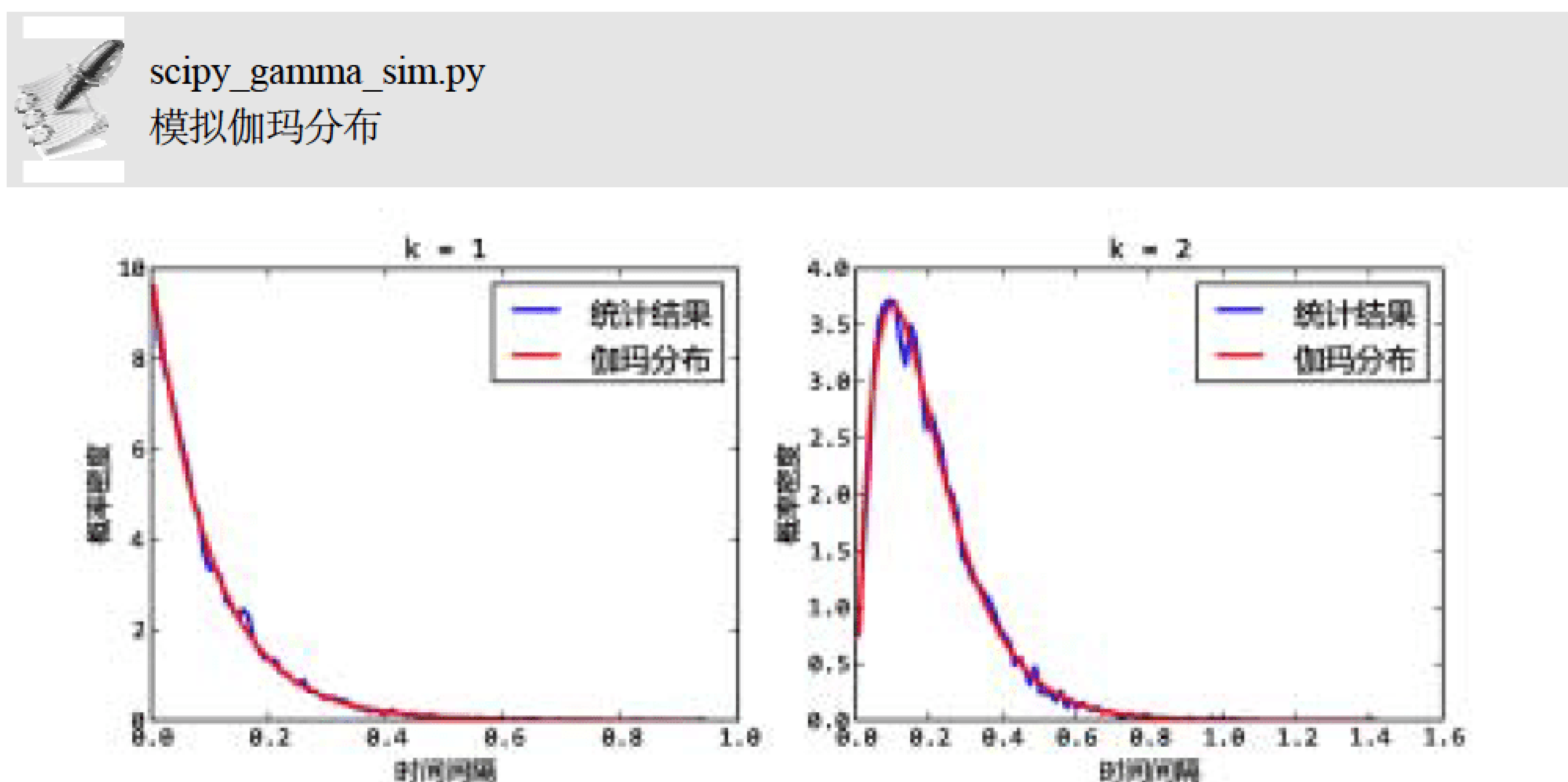


图 3-21　模拟伽玛分布

下面我们直接在 IPython 中模拟伽玛分布。首先在 10 000 秒之内产生 100 000 个随机事件发生的时刻。因此事件的平均发生次数为每秒 10 次：

```
>>> _lambda = 10
>>> time = 10000
>>> t = np.random.rand(_lambda*time)*time
```

为了计算事件前后的时间间隔，需要先对随机时刻进行排序：

```
>>> t.sort()
```

然后分别计算“k=1”和“k=2”时的时间间隔：

```
>>> s1 = t[1:] - t[:-1]      #相邻两个事件之间的时间间隔
>>> s2 = t[2:] - t[:-2]      #相隔一个事件的两个事件之间的时间间隔
```

对 s1 和 s2 分别调用 histogram()进行概率统计，设置 normed 为 True 可以直接统计概率密度：

```
>>> dist1, x1 = np.histogram(s1, bins=100, normed=True)
>>> dist2, x2 = np.histogram(s2, bins=100, normed=True)
```

histogram()返回的第二个值为统计区间的边界，采用 gamma.pdf()计算伽玛分布的概率密度时，使用各个区间的中值进行计算。pdf()的第二个参数为 k 值，scale 参数为 $1/\lambda$：

```
>>> gamma1 = stats.gamma.pdf((x1[:-1]+x1[1:])/2, 1, scale=1.0/_lambda)
>>> gamma2 = stats.gamma.pdf((x2[:-1]+x2[1:])/2, 2, scale=1.0/_lambda)
>>> np.max(np.abs(gamma1 - dist1))
0.13557317865888141
>>> np.max(np.abs(gamma2 - dist2))
0.087375030861794656
```

由于概率密度函数的值本身比较大，因此上面的误差已经很小了：

```
>>> np.max(gamma1), np.max(gamma2)
(9.3483221580498537, 3.6767953241013656)
```

3.8 嵌入 C 语言程序——weave

Python 作为动态语言，功能虽然强大，但它在数值计算方面有一个最大的缺点：速度不够快。在 Python 级别的循环和计算的速度只有 C 语言程序的百分之一。因此才有了 NumPy、SciPy 这样的扩展库，对高度优化的 C、Fortran 的函数库进行封装，以供 Python 程序调用。如果这些高度优化的函数库无法实现我们的算法，就必须从头开始编写循环、进行数值运算，那么用 Python 来做显然是不合适的。因此 SciPy 提供了快速调用 C++语言程序的方法：weave 模块。下面的程序用 weave 模块调用 C 语言对 NumPy 的数组进行求和运算。



scipy_weave_sum_bentchmark.py
用 weave 模块调用 C 语言程序来加快运行速度

```
import scipy.weave as weave
import numpy as np
import time

def my_sum(a):
    n=int(len(a))
    code="""
    int i;
```

```
    double counter;
    counter =0;
    for(i=0;i<n;i++){
        counter=counter+a(i);
    }
    return_val=counter;
    """

    err=weave.inline(
        code,   ❶
        ['a','n'],   ❷
        type_converters=weave.converters.blitz, ❸
        compiler="gcc"  ❹
    )
    return err

a = np.arange(0, 10000000, 1.0)
# 先调用一次 my_sum，weave 会自动对 C 语言进行编译，此后直接运行编译之后的代码
my_sum(a)

start = time.clock()
for i in xrange(100):
    my_sum(a)  # 直接运行编译之后的代码
print "my_sum:", (time.clock() - start) / 100.0

start = time.clock()
for i in xrange(100):
    np.sum( a ) # numpy 中的 sum，其实现也是 C 语言级别
print "np.sum:", (time.clock() - start) / 100.0

start = time.clock()
sum(a) # Python 内部函数 sum 通过数组 a 的迭代接口访问每个元素，因此速度很慢
print "sum:", time.clock() - start
```

此程序在笔者计算机上的运行结果为：

```
my_sum: 0.0311053282622
np.sum: 0.0312218492456
sum: 12.1358849726
```

可以看到，用 weave 模块编译的 C 语言程序和 NumPy 的 sum()几乎一样快，而 Python 的内部函数 sum()使用数组的迭代器接口进行运算，因此速度是 Python 语言级别的，只有 weave 版本的几百分之一。

❶weave.inline()的第一个参数是一个字符串，其内容为需要执行的 C++代码。❷第二个参数是一个列表，它告诉 inline()把 Python 中的变量 a 和 n 传递给 C++程序，注意我们用字符串

表示变量名。❸converters.blitz()是一个类型转换器，将 NumPy 的数组类型转换为 C++的 blitz 类。C++程序中的变量 a 不是一个数组，而是 blitz 对象，因此它使用 a(i)获得各个元素的值，而不是用 a[i]。❹最后我们通过 compiler 参数指定采用 gcc 作为 C++编译器。



如果在编译时出现错误信息，请在命令行中输入"where gcc"来查看 MinGW 编译器的安装路径。如果路径中存在空格或中文，请重新安装 MinGW 编译器到全英文的无空格路径之下。路径中的空格或中文可能会使 Python 无法正确调用它进行编译。

在本书的实战篇，还会对 weave 模块进行详细介绍。这段程序让我们先吃了一颗定心丸：尽管使用 Python 解决你手头上的计算问题，不必在乎计算速度不够快，因为我们可以很容易将需要大量计算的部分使用 C 语言进行改写。

第 4 章

SymPy——符号运算好帮手

SymPy 是 Python 的数学符号计算库，用它可以进行数学表达式的符号推导和演算。本章的实例程序大都是在交互式环境下运行。建议读者使用 isympy 运行本章的演示程序，isympy 在 IPython 的基础上添加了数学表达式的直观显示功能。此外，启动时还会自动运行下面的程序：

```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> k, m, n = symbols('k,m,n', integer=True)
>>> f, g, h = map(Function, 'fgh')
```

这段程序首先将 Python 的除法操作符“/”从整数除法改为普通除法。然后从 SymPy 库载入所有符号，并且定义了三个通用的数学符号 x、y、z，三个表示整数的符号 k、m、n，以及三个表示数学函数的符号 f、g、h。

isympy 的启动程序可以在“C:\Python26\Scripts”下找到。此路径已经添加进了系统路径中，因此可以在任何位置启动它，例如可以使用 Windows 的运行对话框直接运行 isympy。

4.1 从例子开始

在详细介绍 SymPy 的语法结构和各种运算功能之前，让我们先通过两个实例说明用 SymPy 解决符号运算问题的一般步骤。

4.1.1 封面上的经典公式

请读者看一下本书封面左上角的那个数学公式：

$$e^{i\pi}+1=0$$

此公式被称为欧拉恒等式，其中 e 是自然常数，i 是虚数单位，π 是圆周率。此公式被誉为数学中最奇妙的公式，它将 5 个基本数学常数用加法、乘法和幂运算联系起来。下面我们用 SymPy 验证此公式。

从 SymPy 库载入的符号中，E 表示自然常数，I 表示虚数单位，pi 表示圆周率，因此上面的公式可以直接如下计算：

```
>>> E**(I*pi)+1
0
```

SymPy 除了可以直接计算公式的值之外，还可以帮助我们做数学公式的推导和证明。欧拉恒等式可以将 π 代入下面的欧拉公式得到：

$$e^{ix} = \cos x + i \sin x$$

在 SymPy 中可以使用 expand()将表达式展开，我们用它展开 e^{ix} 试试看：

```
>>> expand( E**(I*x) )
exp(I*x)
```

很遗憾没有成功，只是换了一种写法而已。这里的 exp 是 SymPy 中表示自然指数函数的类。当 expand()的 complex 参数为 True 时，表达式将被分为实数和虚数两个部分：

```
>>> expand(exp(I*x), complex=True)
I*exp(-im(x))*sin(re(x)) + cos(re(x))*exp(-im(x))
```

这次将表达式展开了，但是得到的结果相当复杂。其中的 sin、cos、re、im 都是 SymPy 中定义的表示数学函数的类：re 是取实数值的函数，im 是取虚数值的函数。显然，expand()将 x 当做复数了。为了指定 x 为实数，我们需要重新定义 x：

```
>>> x = Symbol("x", real=True)
>>> expand(exp(I*x), complex=True)
I*sin(x) + cos(x)
```

终于得到了我们需要的公式，那么如何证明它呢？我们可以用泰勒多项式对其进行展开：

```
>>> tmp = series(exp(I*x), x, 0, 10)
>>> tmp
```

$$1 + ix - \frac{1}{2}x^2 - \frac{1}{6}ix^3 + \frac{1}{24}x^4 + \frac{1}{120}ix^5 - \frac{1}{720}x^6 - \frac{1}{5040}ix^7 + \frac{1}{40320}x^8 + \frac{1}{362880}ix^9 + o(x^{10})$$

将数学表达式转换为 LaTeX 格式

为了让读者更清晰、直观地查看 SymPy 的计算结果，本书使用数学公式显示较为复杂的结果。幸好 NumPy 提供了 latex()，可以将表达式自动转换为 LaTeX 的数学公式格式，因此这些复杂的数学公式不需要手工输入。例如：

```
>>> latex(exp(I*x))
$e^{\mathbf{\imath} x}$
```

series()对表达式进行泰勒级数展开。我们看到展开之后虚数项和实数项交替出现。根据欧拉公式，虚数项的和应该等于 $\sin x$ 的泰勒展开，而实数项的和应该等于 $\cos x$ 的泰勒展开。

下面获得 tmp 的实部：

```
>>> re(tmp)
```

$$1-\frac{1}{2}x^2+\frac{1}{24}x^4-\frac{1}{720}x^6+\frac{1}{40320}x^8+\mathrm{Re}(o(x^{10}))$$

下面对 cos (x)进行泰勒展开，我们看到其中各项和上面的结果是一致的。

```
>>> series(cos(x), x, 0, 10)
```

$$1-\frac{1}{2}x^2+\frac{1}{24}x^4-\frac{1}{720}x^6+\frac{1}{40320}x^8+o(x^{10})$$

下面获得 tmp 的虚部：

```
>>> im(tmp)
```

$$x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7+\frac{1}{362880}x^9+\mathrm{Im}(o(x^{10}))$$

下面对 sin (x)进行泰勒展开，其中各项也和上面的结果一致。

```
>>> series(sin(x), x, 0, 10)
```

$$x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7+\frac{1}{362880}x^9+o(x^{10})$$

由于 e^{ix} 展开式的实部和虚部分别等于 $\cos x$ 和 $\sin x$，因此验证了欧拉公式的正确性。

4.1.2 球体体积

3.4 节介绍了如何使用数值定积分计算球体的体积，SymPy 中的 integrate()则可以帮助我们进行符号积分。例如下面的语句用 integrate()进行不定积分运算：

```
>>> integrate(x*sin(x), x)
-x*cos(x) + sin(x)
```

如果指定变量 x 的取值范围，integrate()就能进行定积分运算：

```
>>> integrate(x*sin(x), (x, 0, 2*pi))
-2*pi
```

为了计算球体体积，首先看看如何计算圆的面积，假设圆的半径为 r，则圆上任意一点的

Y 坐标函数为：

$$y(x)=\sqrt{r^2-x^2}$$

因此可以直接对函数 $y(x)$在-r 到 r 区间上进行定积分得到半圆面积：

```
>>> x, y, r = symbols('x,y,r')
>>> 2 * integrate(sqrt(r*r-x**2), (x, -r, r))
2*Integral((r**2 - x**2)**(1/2), (x, -r, r))
```

首先需要定义运算中所需的符号，这里用 symbols()一次创建多个符号。integrate()没有计算出积分结果，而是直接返回了我们输入的算式。这是因为 SymPy 不知道 r 是大于 0 的，重新定义 r，就可以得到正确答案了：

```
>>> r = symbols('r', positive=True)
>>> circle_area = 2 * integrate(sqrt(r**2-x**2), (x, -r, r))
>>> circle_area
pi*r**2
```

接下来对此面积公式进行定积分，就可以得到球体的体积，但是随着 X 轴坐标的变化，对应切面的半径也会发生变化。假设 X 轴的坐标为 x，球体的半径为 r，那么 x 处球的切面半径可以使用前面的公式 y(x)计算出。因此需要对圆的面积公式 circle_area 中的变量 r 进行替代：

```
>>> circle_area = circle_area.subs(r, sqrt(r**2-x**2))
>>> circle_area
pi*(r**2 - x**2)
```

用 subs 进行算式替换

subs()可以将算式中的符号进行替换，它有 3 种调用方式：

- `expression.subs(x, y)`：将算式中的 x 替换成 y。
- `expression.subs({x:y,u:v})`：使用字典进行多次替换。
- `expression.subs([(x,y),(u,v)])`：使用列表进行多次替换。

请注意多次替换是顺序执行的，因此：

```
expression.sub([(x,y),(y,x)])
```

并不能对符号 x 和 y 进行交换。

然后对 circle_area 中的变量 x 在区间-r 到 r 上进行定积分，就可以得到球体的体积公式：

```
>>> integrate(circle_area, (x, -r, r))
4*pi*r**3/3
```

4.2　数学表达式

本节详细介绍数学表达式的结构，虽然这部分内容比较枯燥，但只有了解表达式的结构，才能随心所欲地对其进行处理，将 SymPy 运用到更复杂的计算中。

4.2.1　符号

数学符号用 Symbol 对象表示，符号对象的 name 属性是符号名，符号名在显示由此符号构成的表达式时使用。Symbol 对象和 Python 的变量没有内在联系，但是为了使用起来方便，通常我们让变量名和符号名相同。为了快速创建符号及与其同名的变量，可以使用 var()，例如：

```
>>> var("x0,y0,x1,y1")
(x0, y0, x1, y1)
```

上面的语句创建了名为 x0、y0、x1、y1 的 4 个 Symbol 对象，同时还在当前的环境中创建了 4 个同名的变量来分别表示这 4 个 Symbol 对象。因为符号对象在转换为字符串时直接使用它的 name 属性，因此在交互式环境中我们看到变量 x0 的值就是 x0，但是查看变量 x0 的类型时就可以发现，它实际上是一个 Symbol 对象。

```
>>> x0
x0
>>> type(x0)
<class 'sympy.core.symbol.Symbol'>
>>> x0.name
x0
>>> type(x0.name)
<type 'str'>
```

在交互式环境中使用 var()能够快速创建变量和 Symbol 对象，但是在程序中使用它容易引起混淆，这时我们可以使用 symbols()创建 Symbol 对象，再将它们赋值给变量：

```
>>> x1,y1 = symbols("x1,y1")
>>> type(x1)
<class 'sympy.core.symbol.Symbol'>
```

当然，如果不嫌麻烦也可以直接使用 Symbol 类创建对象：

```
>>> x2 = Symbol("x2")
```



当 symbols()的符号名参数中没有逗号或空格时，它将为每个字符创建一个 Symbol 对象。例如 symbols("abc")将创建名称分别为 a、b、c 的三个 Symbol 对象。此功能在后续版本中可能会被删除。

变量名和符号名当然也可以是不一样的，例如：

```
>>> t = x0
>>> t
x0
>>> a,b = symbols("alpha,beta")
>>> a, b
(alpha, beta)
```

数学公式中的符号一般都有特定的假设，例如 m、n 通常是整数，而 z 经常表示复数。在用 var()、symbols()或 Symbol()创建 Symbol 对象时，可以通过关键字参数指定所创建符号的假设条件，这些假设条件会影响到它们所参与的计算。例如，下面创建了两个整数符号 m 和 n，以及一个正数符号 x：

```
>>> m, n = symbols("m,n", integer=True)
>>> x = Symbol("x", positive=True)
```

每个符号都有许多 is_*属性，用以判断符号的各种假设条件。在 IPython 中，使用自动完成功能可以快速查看这些假设的名称。注意下划线后为大写字母的属性，用来判断对象的类型；而全小写字母的属性，则用来判断符号的假设条件。

```
>>> x.is_ # 按 Tab 键自动完成
x.is_Add             x.is_bounded          x.is_nonnegative
x.is_Atom            x.is_commutative      x.is_nonpositive
x.is_Derivative      x.is_comparable       x.is_nonzero
x.is_Function        x.is_complex          x.is_number
x.is_Integer         x.is_composite        x.is_odd
x.is_Mul             x.is_even             x.is_polynomial
x.is_Number          x.is_finite           x.is_positive
x.is_NumberSymbol    x.is_hypergeometric   x.is_prime
x.is_Order           x.is_imaginary        x.is_rational
x.is_Piecewise       x.is_infinitesimal    x.is_rational_function
x.is_Pow             x.is_integer          x.is_real
x.is_Rational        x.is_irrational       x.is_unbounded
x.is_Real            x.is_negative         x.is_zero
x.is_Symbol          x.is_noninteger
>>> x.is_Symbol # x 是一个符号
True
>>> x.is_positive # x 是一个正数
True
>>> x.is_imaginary # 因为 x 可以比较大小，所以它不是虚数
False
>>> x.is_complex # x 是一个复数，因为复数包括实数，而实数包括正数
True
```

使用 assumptions0 属性可以快速查看所有的假设条件，其中 commutative 为 True 表示此符号满足交换律，其余的假设条件根据英文名很容易知道它们的含义，这里就不再详细叙述了。

```
>>> x.assumptions0
{commutative: True,
 complex: True,
 imaginary: False,
 negative: False,
 nonnegative: True,
 nonpositive: False,
 nonzero: True,
 positive: True,
 real: True,
 zero: False}
```

在 SymPy 中，所有的对象都从 Basic 类继承，实际上这些 is_*属性和 assumptions0 属性都是在 Basic 类中定义的：

```
>>> Symbol.mro()
[<class 'sympy.core.symbol.Symbol'>,
 <class 'sympy.core.basic.Atom'>,
 <class 'sympy.core.basic.Basic'>,
 <class 'sympy.core.assumptions.AssumeMeths'>,
 <type'object'>]
```

4.2.2 数值

为了实现符号运算，在 SymPy 内部有一整套数值运算系统。因此 SymPy 的数值和 Python 的整数、浮点数是完全不同的对象。为了使用方便，SymPy 会尽量自动将 Python 的数值类型转换为 SymPy 的数值类型。此外，SymPy 提供了一个 S 对象用于进行这种转换。在下面的例子中，当有 SymPy 的数值参与计算时，结果将是 SymPy 的数值对象。

```
>>> 1/2 + 1/3 # 结果为浮点数
0.833333333333
>>> S(1)/2 + 1/S(3) # 结果为 SymPy 的数值对象
5/6
```

“5/6”在 SymPy 中使用 Rational 对象表示，它由两个整数的商表示，数学上称之为有理数。也可以直接通过 Rational 创建：

```
>>> Rational(5, 10) # 有理数会自动进行约分处理
1/2
```

实数用 Real 对象表示，它和标准的浮点数类似，但是它的精度(有效数字)可以通过参数指定。由于在浮点数和 Real 对象内部都使用二进制的方式表示数值，因此它们都无法精确表示十进制中的精确小数，例如 0.1。我们可以使用 N()函数查看浮点数的实际数值，例如下面的语句查看浮点数 0.1 和 10000.1 的 60 位有效数字，可以看到数值的绝对精度随着数值的增大而减小：

```
>>> N(0.1, 60)
0.1000000000000000055511151231257827021181583404541015625000000
>>> N(10000.1, 60)
10000.1000000000003637978807091712951660156250000000000000000
```

由于浮点数的精度有限，因此使用它创建 Real 对象时，即使指定精度参数也不能缩小它与理想值之间的误差，这时我们可以使用字符串表示数值：

```
>>> N(Real(0.1,60),60) #用浮点数创建 Real 对象时，精度和浮点数相同
0.1000000000000000055511151231257827021181583404541015625000000
>>> N(Real("0.1",60),60) #用字符串创建 Real 对象时，所指定的精度有效
0.100000000000000000000000000000000000000000000000000000000000
>>> N(Real("0.1",60),65) #提高表示精度，就会发现它也不是完全精确的
0.099999999999999999999999999999999999999999999999999999999996111
```

4.2.3　运算符和函数

SymPy 重新定义了所有的数学运算符和数学函数。例如：Add 类表示加法，Mul 类表示乘法，Pow 类表示指数运算，sin 类表示正弦函数。和 Symbol 对象一样，这些运算符和函数都从 Basic 类继承，请读者自行在 IPython 中查看它们的继承列表(例如 Add.mro())。可以使用这些类创建复杂的表达式：

```
>>> var("x,y,z,n")
>>> Add(x,y,z)
x + y + z
>>> Add(Mul(x,y,z), Pow(x,y), sin(z))
x*y*z + sin(z) + x**y
```

由于在 Basic 类中重新定义了__add__()等用于创建表达式的方法，因此可以使用和 Python 表达式相同的方式创建 SymPy 的表达式：

```
>>> x*y*z + sin(z) + x**y
x*y*z + sin(z) + x**y
```

Basic 类中定义了两个很重要的属性：func 和 args。func 属性得到对象的类，而 args 得到其参数。使用这两个属性可以观察 SymPy 所创建的表达式。也许读者会对没有减法运算类感到奇怪，下面让我们看看减法运算所得到的表达式：

```
>>> t = x - y
>>> t.func # 减法运算用加法类 Add 表示
<class 'sympy.core.add.Add'>
>>> t.args # 两个加数一个是 x，一个是-y
(x, -y)
>>> t.args[1].func # -y 是用 Mul 表示的
<class 'sympy.core.mul.Mul'>
>>> t.args[1].args
(-1, y)
```

通过上面的例子可以看出，表达式“x-y”在 SymPy 中实际上是用“Add(x, Mul(-1, y))”表示的。同样，SymPy 中没有除法类，请读者使用和上面相同的方法观察“x/y”在 SymPy 中是如何表示的。

SymPy 的表达式实际上是一个由 Basic 类的各种对象进行多层嵌套所得到的树状结构。下面的函数使用递归显示这种树状结构：



Print_expression.py

显示 SymPy 的表达式

```
def print_expression(e, level=0):
    spaces = "    "*level
    if isinstance(e, (Symbol, Number)):
        print spaces + str(e)
        return
    if len(e.args) > 0:
        print spaces + e.func.__name__
        for arg in e.args:
            print_expression(arg, level+1)
    else:
        print spaces + e.func.__name__
```

例如 $\sqrt{x^2+y^2}$ 在 SymPy 中使用下面的树表示：

```
>>> print_expression(sqrt(x**2+y**2))
Pow
    Add
        Pow
            y
            2
        Pow
            x
            2
    1/2
```

由于其中各个对象的 args 属性类型是元组，因此表达式一旦创建就不能再改变。使用不可变的结构表示表达式有很多优点，例如可以将表达式作为字典的键。

除了使用 SymPy 中预先定义好的具有特殊运算含义的数学函数之外，还可以使用 Function() 创建自定义的数学函数：

```
>>> f = Function("f")
```

请注意 Function 虽然是一个类，但是上面的语句所得到的 f 并不是 Function 类的实例。和预定义的数学函数一样，f 是一个类，它从 Function 类继承：

```
>>> f.__base__
Function
>>> isinstance(f, Function)
False
```

在 Python 中，类型(类)也是对象，通常类型对象的类型是 type 对象，通过 type()可以获得对象的类型，因此：

```
>>> type([1,2,3]) # 列表对象的类型是列表
<type 'list'>
>>> type(list) # 列表类型的类型是 type
<type 'type'>
```

但是通过 Function 创建的类型对象 f 的类型并不是 type：

```
>>> type(f)
<class 'sympy.core.function.FunctionClass'>
```

当使用 f 创建一个表达式时，就相当于创建它的一个实例：

```
>>> t = f(x,y)
>>> type(t)
f
>>> t.func
f
>>> t.args
(x, y)
```

f 的实例 t 可以参与表达式运算：

```
>>> t+t*t
```

$$f(x,y)+f^2(x,y)$$

4.3 符号运算

SymPy 提供的符号运算功能十分丰富，然而由于篇幅所限，本节只能对 SymPy 中一些常用的符号运算功能进行简单的介绍。

4.3.1 表达式变换和化简

simplify()可以对数学表达式进行化简，例如：

```
>>> simplify((x+2)**2 - (x+1)**2)
3 + 2*x
```

simplify()调用 SymPy 内部的多种表达式变换函数对表达式进行化简运算。但是数学表达式的化简是一件非常复杂的工作，并且对于同一个表达式，根据其使用目的可以有多种化简方案。因此本节介绍 SymPy 提供的各种表达式变换函数，充分利用这些函数可以实现表达式的变换和化简。

radsimp()对表达式的分母进行有理化，它所得到的表达式的分母部分将不含无理数。例如：

```
>>> radsimp(1/(sqrt(5)+2*sqrt(2)))
```

$$-\frac{1}{3}\sqrt{5}+\frac{2}{3}\sqrt{2}$$

它也可以对带符号的表达式进行处理：

```
>>> radsimp(1/(y*sqrt(x)+x*sqrt(y)))
```

$$\frac{x\sqrt{y}-y\sqrt{x}}{yx^2-xy^2}$$

ratsimp()对表达式中的分母进行通分运算，即将表达式转换为分子除分母的形式：

```
>>> ratsimp(x/(x+y)+y/(x-y))
```

$$\frac{x(x-y)+y(x+y)}{(x-y)(x+y)}$$

fraction()返回一个包含表达式的分子和分母的元组，用它可以获得 ratsimp()通分之后的分子或分母：

```
>>> fraction(ratsimp(1/x+1/y))
(x + y, x*y)
```

请注意 fraction()不会自动对表达式进行通分运算，因此：

```
>>> fraction(1/x+1/y)
(1/x + 1/y, 1)
```

cancel()和 trim()对分式表达式的分子分母进行约分运算，去除它们的公因式。cancel()只能对纯符号的分式表达式进行约分；而 trim()则可以对表达式中的任意部分进行约分，并且支持函数表达式的约分运算：

```
>>> cancel((x**2-1)/(1+x))
-1 + x
>>> cancel(sin((x**2-1)/(1+x))) # cancel 不能对函数内部的表达式进行约分
sin((x**2-1)/(1+x))
>>> trim(sin((x**2-1)/(1+x))) # trim 可以对内部表达式进行约分
-sin(1 - x)
>>> cancel((f(x)**2-1)/(f(x)+1)) # cancel 不能对带函数的表达式进行约分
(f(x)**2-1)/(f(x)+1)
>>> trim((f(x)**2-1)/(f(x)+1)) # trim 可以对带函数的表达式进行约分
-1 + f(x)
```

trigsimp()对表达式中的三角函数进行化简。它有两个可选参数——deep 和 recursive，默认值都为 False。当 deep 参数为 True 时，将对表达式中的所有子表达式进行简化运算；当 recursive 参数为 True 时，将递归使用 trigsimp()进行最大限度的化简：

```
>>> trigsimp(sin(x)**2+2*sin(x)*cos(x)+cos(x)**2)
1 + 2*cos(x)*sin(x)
>>> trigsimp(f(sin(x)**2+2*sin(x)*cos(x)+cos(x)**2)) # 对于嵌套的表达式不进行简化
f(2*cos(x)*sin(x) + cos(x)**2 + sin (x)**2)
>>> trigsimp(f(sin(x)**2+2*sin(x)*cos(x)+cos(x)**2), deep=True) # 对嵌套表达式进行处理
f(1 + 2*cos(x)*sin(x))
```

请注意，虽然$2\cos x\sin x$还可以再简化为$\sin 2x$，但是目前 trigsimp()对这种简化无能为力。

expand_trig()可以对三角函数的表达式进行展开。它实际上是对 expand()的封装，通过将 expand()的 trig 参数设置为 True，实现三角函数的展开计算。读者可以在 IPython 下输入“expand_trig??”来查看它调用 expand()时的参数。

```
>>> expand_trig(sin(2*x+y))
```

$$-\sin(y)+2\cos^2(x)\sin(y)+2\cos(x)\cos(y)\sin(x)$$

expand()根据用户设置的标志参数对表达式进行展开。默认情况下，以下的标志参数为 True。

mul：展开乘法

```
>>> expand(x*(y+z))
```

```
x*y + x*z
```

log：展开对数函数参数中的乘积和幂运算

```
>>> x,y=symbols("x,y",positive=True)
>>> expand(log(x*y**2))
2*log(y) + log(x)
```

multinomial：展开加法式的整数次幂

```
>>> expand((x+y)**3)
3*x*y**2 + 3*y*x**2 + x**3 + y**3
```

power_base：展开幂函数的底数乘积

```
>>> expand((x*y)**z)
x**z*y**z
```

power_exp：展开幂函数的指数和

```
>>> expand(x**(y+z))
x**y*x**z
```

可以将默认为 True 的标志参数设置为 False，强制不展开对应的表达式。在下面的例子中，将 mul 设置为 False，因此不对乘法进行展开：

```
>>> x,y,z=symbols("x,y,z", positive=True)
>>> expand(x*log(y*z), mul=False)
x*(log(y) + log(z))
```

expand()的以下标志参数默认为 False。

complex：展开复数的实部和虚部

```
>>> x,y=symbols("x,y",complex=True)
>>> expand(x*y, complex=True)
re(x)*re(y) - im(x)*im(y) + I*im(x)*re(y) + I*im(y)*re(x)
```

func：对一些特殊函数进行展开

```
>>> expand(gamma(1+x),func=True)
x*gamma(x)
```

trig：展开三角函数

```
>>> expand(sin(x+y), trig=True)
cos(x)*sin(y) + cos(y)*sin(x)
```

expand_log()、expand_mul()、expand_complex()、expand_trig()、expand_func()等函数则通过将相应的标志参数设置为 True，对 expand()进行封装。

factor()可以对多项式表达式进行因式分解：

```
>>> factor(15*x**2+2*y-3*x-10*x*y)
(1 - 5*x)*(-3*x + 2*y)
>>> factor(expand((x+y)**20))
(x+y)**20
```

collect()收集表达式中指定符号的有理指数次幂的系数。例如，我们希望获得如下表达式中 x 的各次幂的系数：

```
>>> eq = (1+a*x)**3 + (1+b*x)**2
```

首先需要对表达式 eq 进行展开，得到的表达式 eq2 是一系列乘式的和：

```
>>> eq2 = expand(eq)
>>> eq2
```

$$2+2bx+3ax+b^2x^2+3a^2x^2+a^3x^3$$

然后调用 collect()，对表达式 eq2 中 x 的幂的系数进行收集：

```
>>> collect(eq2, x)
```

$$2+x(2b+3a)+x^2(b^2+3a^2)+a^3x^3$$

默认情况下，collect()返回的是一个整理之后的表达式，如果我们希望得到 x 的各次幂的系数，可以设置 evaluate 参数为 False，让它返回一个以 x 的幂为键、值为系数的字典：

```
>>> p = collect(eq2, x, evaluate=False)
>>> p[S(1)] # 常数项，注意需要用 SymPy 中的数值 1，或者使用 p[x**0]
2
>>> p[x**2] # x 的 2 次项系数
b**2 + 3*a**2
```

collect()也可以收集表达式的各次幂的系数，例如下面的程序收集表达式“sin(2*x)”的系数：

```
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)
```

4.3.2 方程

在 SymPy 中，表达式可以直接表示值为 0 的方程。也可以使用 Eq()创建方程。solve()可以对方程进行符号求解，它的第一个参数是表示方程的表达式，其后的参数是表示方程中未知变

量的符号。下面的例子使用 solve()对一元二次方程进行求解：

```
>>> a,b,c = symbols("a,b,c")
>>> solve(a*x**2+b*x+c, x)
```

$$\left[-\frac{b+\sqrt{-4ac+b^2}}{2a}, \frac{-b+\sqrt{-4ac+b^2}}{2a}\right]$$

由于方程的解可能有多组，因此 solve()返回一个列表保存所有的解。可以传递包含多个表达式的元组或列表，让 solve()对方程组进行求解，得到的解是两层嵌套的列表，其中每个元组表示方程组的一组解：

```
>>> solve((x**2+x*y+1,y**2+x*y+2),x,y)
```

$$\left[(\frac{1}{3}i\sqrt{3},\frac{2}{3}i\sqrt{3}),(-\frac{1}{3}i\sqrt{3},-\frac{2}{3}i\sqrt{3})\right]$$

4.3.3 微分

Derivative 是表示导函数的类，它的第一个参数是需要进行求导的数学函数，第二个参数是求导的自变量。请注意 Derivative 所得到的是一个导函数，它并不会进行求导运算：

```
>>> t = Derivative(sin(x), x)
>>> t
D(sin(x), x)
```

如果希望它进行实际的运算，计算出导函数，可以调用其 doit()方法：

```
>>> t.doit()
cos(x)
```

也可以直接使用 diff()函数或表达式的 diff()方法来计算导函数：

```
>>> diff(sin(2*x), x)
2*cos(2*x)
>>> sin(2*x).diff(x)
2*cos(2*x)
```

使用 Derivative 对象可以表示我们自定义的数学函数的导函数，例如：

```
>>> Derivative(f(x), x)
D(f(x), x)
```

由于 SymPy 不知道如何对自定义的数学函数进行求导，因此它的 diff()方法会返回和上面

相同的结果：

```
>>> f(x).diff(x)
D(f(x), x)
```

添加更多的符号参数可以表示高阶导函数，例如：

```
>>> Derivative(f(x), x, x, x) # 也可以写作 Derivative(f(x), x, 3)
```

$$\frac{\partial^2}{\partial^3 x} f(x)$$

也可以表示多个变量的导函数，例如：

```
>>> Derivative(f(x,y), x,2,y,3)
```

$$\frac{\partial^5}{\partial^2 x \partial^3 y} f(x, y)$$

diff()的参数和 Derivative 相同，例如下面的语句计算 $\sin(xy)$ 对 x 两次求导、对 y 三次求导的结果：

```
>>> diff(sin(x*y), x,2,y,3)
```

$$-6x\cos(xy) + x^3 y^2 \cos(xy) + 6yx^2 \sin(xy)$$

4.3.4 微分方程

dsolve()可以对微分方程进行符号求解。它的第一个参数是一个带未知函数的表达式，第二个参数是需要进行求解的未知函数。例如下面的程序对微分方程 $f'(x) - f(x) = 0$ 进行求解。得到的结果是一个自然指数函数，它有一个待定系数 c_1。

```
>>> f=Function("f")
>>> dsolve(Derivative(f(x),x) - f(x), f(x))
```

$$f(x) = e^{c_1 + x}$$

用 dsolve()解微分方程时可以传递一个 hint 参数，指定微分方程的解法。该参数的默认值为"default"，表示由 SymPy 自动挑选解法。可以将 hint 参数设置为"best"，让 dsolve()尝试所有已知解法，并返回最简单的解，例如下面对微分方程 $\frac{\partial}{\partial x} f(x) + f(x) + f^2(x) = 0$ 进行求解。得到的结果是一个一般方程，它描述了 $f(x)$ 和自变量之间的关系。一般把这种函数称为隐函数：

```
>>> x = symbols("x", real=True) # 定义符号 x 为实数
>>> eq1 = dsolve(f(x).diff(x) + f(x)**2 + f(x), f(x))
```

```
>>> eq1
-log(1 + f(x)) + log(f(x)) = C1 - x
```

如果设置 hint 参数为"best"，就能得到更简单的显函数表达式：

```
>>> eq2 = dsolve(f(x).diff(x) + f(x)**2 + f(x), f(x), hint="best")
>>> eq2
```

$$f(x)=\frac{e^{-x}}{c_1-e^{-x}}$$

下面让我们用程序验证一下上面的两个结果是等价的。eq1 和 eq2 是两个表示等式的 Equality 对象，等号左右两边的表达式可以通过其 lhs 和 rhs 属性获得：

```
>>> eq1.func
<class 'sympy.core.relational.Equality'>
>>> eq1.lhs
-log(1 + f(x)) + log(f(x))
>>> eq1.rhs
C1 - x
```

为了验证 eq1 和 eq2 两个等式是等价的，只需要将 eq2 代入到 eq1 中进行化简即可。这个替换工作可以由 subs()完成，将 subs()得到的结果经由 simplify()进行化简：

```
>>> simplify(eq1.lhs.subs({f(x):eq2.rhs}))
-x - log(C1)
```

可以看到除了常数项之外，上面的结果和 eq1 的右式是一样的。

4.3.5 积分

integrate()可以计算定积分和不定积分：

- integrate(f, x)：计算不定积分 $\int f dx$
- integrate(f, (x,a,b))：计算定积分 $\int_a^b f dx$

如果要对多个变量计算多重积分，只需要将被积分的变量依次列出即可：

- integrate(f, x, y)：计算双重不定积分 $\iint f dx dy$
- integrate(f, (x,a,b), (y,c,d))：计算双重定积分 $\int_c^d\int_a^b f dx dy$

和 Derivative 对象表示微分表达式类似，Integral 对象表示积分表达式，它的参数和 integrate()类似，例如：

```
>>> e = Integral(x*sin(x), x)
>>> e
```

$$\int x\sin(x)dx$$

调用积分对象的 doit()方法可以对其进行求值计算：

```
>>> e.doit()
-x*cos(x) + sin(x)
```

有些积分表达式无法进行符号化简，这时可以调用其 evalf()方法或用求值函数 N()对其进行数值运算：

```
>>> e2 = Integral(sin(x)/x, (x, 0, 1))
>>> e2.doit()
```

$$\int_0^2 \frac{\sin(x)}{x}dx$$

由于无法进行符号定积分，因此 doit()返回积分表达式本身。下面我们用 evalf()和 N()对其进行数值运算：

```
>>> e2.evalf()
0.946083070367183
>>> N(e2)
0.946083070367183
>>> N(e2, 100) # 可以指定精度
0.946083070367183014941353313823179657812337954738111
```

实际上$\frac{\sin(x)}{x}$的积分被定义为一个特殊函数，它从 0 到无穷的定积分为$\pi/2$，即：

$$\int_0^\infty \frac{\sin(x)}{x}dx = \pi/2$$

SymPy 的数值计算功能还不够强大，不能对应如下这种情况的无限积分：

```
>>> N(Integral(sin(x)/x, (x, 0, oo))) # oo 表示正无穷
.0e+0
```

将积分上限修改为 10000 也没能计算出近似结果，上限为 1000 时得到了$\pi/2$的近似值，不过还远远不够精确：

```
>>> N(Integral(sin(x)/x, (x, 0, 10000)))
.0e+0
>>>N(Integral(sin(x)/x, (x, 0, 1000)))
1.57023312196877
```

as_sum()方法可以将定积分转换为近似求和公式，它将积分区域分割成 N 个小矩形的面积之和：

```
>>> Integral(sin(x), (x, 0, 1)).as_sum(5)
```

$$\frac{1}{5}\sin(\frac{1}{2})+\frac{1}{5}\sin(\frac{1}{10})+\frac{1}{5}\sin(\frac{3}{10})+\frac{1}{5}\sin(\frac{7}{10})+\frac{1}{5}\sin(\frac{9}{10})$$

4.4 其他功能

4.4.1 平面几何

使用 SymPy 的 geometry 模块可以创建表示二维几何图形的对象，例如直线、线段、圆等，并计算这些对象的各种信息。例如计算椭圆的面积，判断一组点是否共线，或者求两条直线的交点。下面我们通过一个例子介绍 geometry 模块的基本用法。

如图 4-1 所示，D 是三角形 ABC 的内切圆圆心，过 C、D、B 三点的圆与直线 AB 和 AC 分别交于 F 和 G 两点。线段 i 和 j 是交点构成的弦，可以证明它们的长度是相等的。用 geometry 模块无法帮助我们证明这个命题，但是可以通过计算线段 i 和 j 的长度对其进行验证。

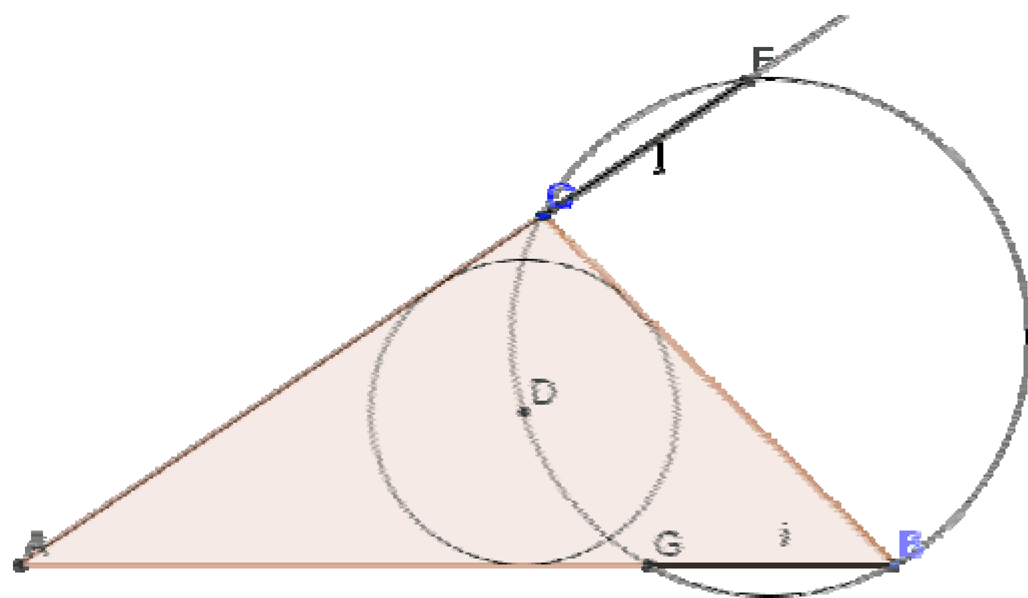


图 4-1　弦相等几何题示意图



sympy_geometry.py
验证弦相等

geometry 模块用解析几何的形式表示几何实体，因此需要用坐标表示点的位置。虽然三角形的顶点的位置是任意的，但是我们可以让点 A 处于原点、点 B 在 X 轴上、点 C 在 Y 轴的上

方，例如：

```
>>> A = Point(0,0)
>>> B = Point(5,0)
>>> C = Point(3,2)
```

Point 对象表示坐标点。其中的 5、3、2 等数值可以换成任意其他的值，只要 A、B、C 三点不共线，能构成三角形即可。下面创建以这三个点为顶点的三角形 t：

```
>>> t = Triangle(A,B,C)
```

Triangle 对象表示三角形。三角形的内切圆圆心可以通过其 incenter 属性获得：

```
>>> D =t.incenter
>>> D
```

$$\left(\frac{140-30\sqrt{2}-10\sqrt{26}+40\sqrt{13}}{30+10\sqrt{13}},\frac{50-20\sqrt{2}+10\sqrt{13}}{30+10\sqrt{13}}\right)$$

虽然三角形的顶点坐标是整数，但是它的内切圆圆心的坐标却很复杂。目前 geometry 模块[①]只能对精确数值进行计算，无法使用浮点数。因此当几何图形十分复杂时，计算量将非常大，希望它以后能支持浮点数，进行更复杂的几何运算。

Circle 对象表示圆。可以通过指定圆心坐标、半径，或者圆上的三个点来创建。这里用 C、D、B 三点创建圆 p：

```
>>> p = Circle(C,D,B)
```

圆和直线的交点可以通过圆的 intersection()方法计算：

```
>>> i = Segment(*p.intersection(Line(A,B)))
>>> j = Segment(*p.intersection(Line(A,C)))
```

这里的 Line(A,B)和 Line(A,C)是临时创建的直线对象。由于圆和直线有两个交点，因此 intersection()方法返回的是一个有两个 Point 对象的列表。通过这两个交点创建表示弦线段的 Segment 对象。

接下来通过 Segment 对象的 length 属性查看其长度。注意 length 属性是一个表示精确值的表达式，它非常复杂，因此我们调用它的 evalf()查看其近似的浮点值：

```
>>> i.length.evalf(50)
1.3944487245360107068807787325295040537487034261548
>>> j.length.evalf(50)
1.3944487245360107068807787325295040537487034261548
```

① 本书写作时 SymPy 的版本号为 0.6.7。

4.4.2 绘图

用 Plot()可以快速绘制二维函数曲线或三维函数曲面。



如果调用绘图函数时出现“Window initialization failed”错误，请按照下面的说明进行修改。

为操作系统添加一个环境变量：“PYGLET_SHADOW_WINDOW=0”。或者在程序开头添加下面的代码，为程序的运行环境临时添加环境变量：

```
import os
os.environ['PYGLET_SHADOW_WINDOW']="0"
```



sympy_plot2d.py
用 SymPy 绘制二维函数曲线

Plot()的参数十分灵活，我们直接通过实例进行讲解。在 IPython 交互环境下运行绘图代码之前，请先运行上面的代码以添加环境变量。

```
>>> p = Plot(x/2)
```

将弹出一个名为“SymPy Plot”的绘图窗口，并绘制一条方程为 $y = x / 2$ 的直线。Plot()返回的是一个 Plot 对象：

```
>>> type(p)
<class 'sympy.plotting.plot.Plot'>
```

Plot 对象可以当做列表使用，其中的每个元素都表示图表中的一个绘图实体，由于前面只绘制了一条曲线，因此对象 p 的长度为 1，图中的曲线与一个 Cartesian2D 对象对应。

```
>>> len(p)
1
>>> p[0] # 图中的曲线
x/2, [x,-5,5,100], 'mode=cartesian; color=rainbow; style=solid'
>>> type(p[0]) # 曲线是一个 Cartesian2D 对象
<class 'sympy.plotting.plot_modes.Cartesian2D'>
>>> p # 直接查看图表中的元素列表
[0]: x/2, 'mode=cartesian'
```

请注意曲线表达式和符号名没有关系，它只和符号的个数有关，表达式中只有一个符号时，将绘制二维曲线。下面再添加几条曲线，直接调用 append()将曲线所对应的函数表达式添加进绘图对象即可：

```
>>> p.append(x**2)
>>> p.append(log(x))
>>> p
[0]: x/2, 'mode=cartesian'
[1]: x**2, 'mode=cartesian'
[2]: log(x), 'mode=cartesian'
```

当传递两个带同一符号的表达式给 Plot()时，将绘制二维参数曲线：

```
>>> p.append(sin(x), cos(x)) # 添加一个圆
>>> p[3]
sin(x), cos(x), 'mode=parametric'
```

也可以通过参数指定在极坐标系中绘图，请注意所有的参数是通过一个分号隔开的字符串表示，而不是通过函数的关键字参数传递。在下面的例子中，设置"mode"绘图参数为"polar"，因此将在极坐标中绘制函数$r = \cos(2\theta)$：

```
>>> p.append(cos(2*x),"mode=polar")
```

Plot 对象的 axes 属性是表示坐标轴的 PlotAxes 对象，通过设置它的属性可以改变坐标轴的一些显示效果，例如下面的语句显示坐标轴上的刻度的数值：

```
>>> p.axes._label_axes = True
```

最后调用 p.saveimage()将图表保存成图像文件，结果如图 4-2 所示(见文前彩插)。

```
>>> p.saveimage("sympy_plot2d.png")
```



显示刻度值会极大影响图表的绘制速度，因此建议只在保存图像时显示刻度线。

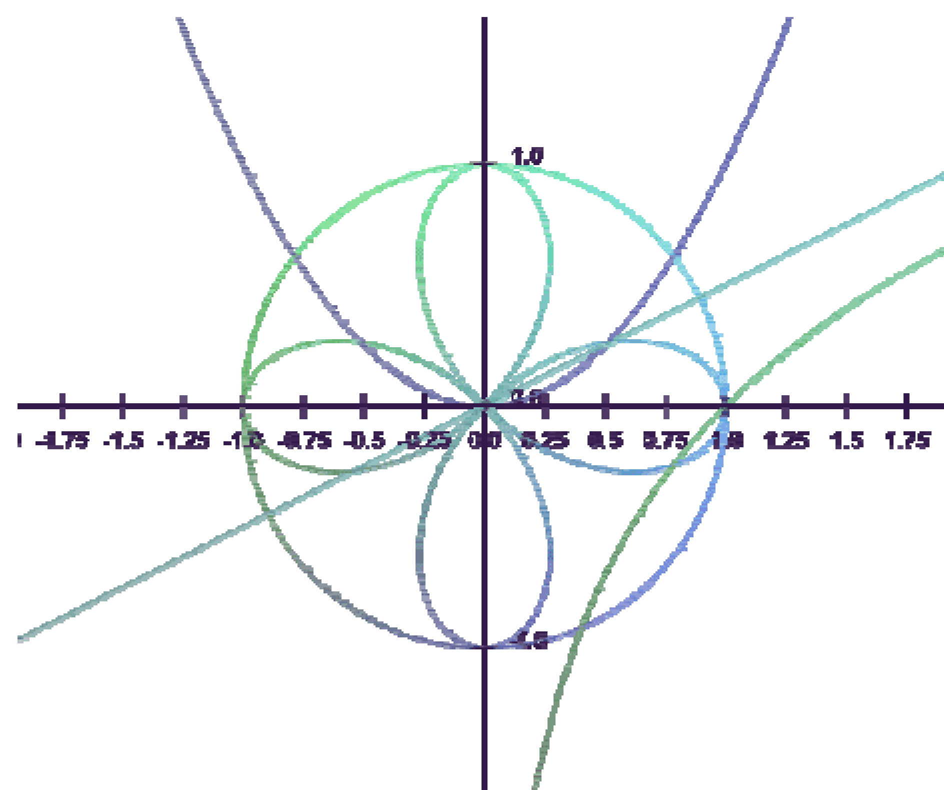


图 4-2　使用 SymPy 绘制的函数曲线

Plot()还可以在三维空间中绘图，下面列出 SymPy 支持的三维绘图模式：

- 双变量表达式，当"mode"参数默认时，在直角坐标系中绘制曲面，默认自变量名为 x 和 y。
- 双变量表达式，"mode=cylindrical"，在圆柱坐标系中绘制曲面，默认自变量名为 t 和 h。
- 双变量表达式，"mode=spherical"，在球坐标系中绘制曲面，默认自变量名为 t 和 p。
- 三个单变量表达式，绘制三维空间中的参数曲线，默认自变量名为 t。
- 三个双变量表达式，绘制三维空间中的参数曲面，默认自变量名为 u 和 v。

由于三维空间中的曲面需要两个变量才能确定，为了不混淆变量含义，建议使用默认的自变量名创建表达式，或者通过指定自变量范围来明确每个变量的含义。例如在圆柱坐标系中，变量 t 表示高度，h 表示角度，而表达式计算的值是半径。当表达式中缺少自变量时，通过指定自变量范围可以明确指定其含义。第一个指定范围的自变量为角度，第二个为高度。

例如，下面在圆柱坐标系中绘制一个 Z 轴方向的正弦波绕 Z 轴旋转 180° 所得的旋转面，效果如图 4-3(左)所示(见文前彩插)。

```
>>> t, h = symbols("t,h")
>>> Plot(sin(t)+2, [h,0,pi],[t,0,2*pi], "mode=cylindrical")
```

由于表达式中缺少变量 h，因此通过第二个参数指定旋转角度变量 h 的范围为 0° 到 180°。而 Z 轴方向正弦波的变量范围是 0° 到 360°。

最后看一个用参数曲面绘制麦比乌斯带的例子，效果如图 4-3(右)所示(见文前彩插)。

用一个纸带旋转半圈再把两端粘上之后所形成的曲面只有一个表面和一个边界，这种曲面称为麦比乌斯带。它可以用如下参数方程描述：

$$x(u,v)=\left(1+\frac{1}{2}v\cos\frac{1}{2}u\right)\cos u$$

$$y(u,v)=\left(1+\frac{1}{2}v\cos\frac{1}{2}u\right)\sin u$$

$$z(u,v)=\frac{1}{2}v\sin\frac{1}{2}u$$

其中变量 u 的取值范围是 0 到 2π，变量 v 的取值范围是-1 到 1。下面是完整的源程序。当传递三个双变量表达式时，Plot()将绘制参数曲面。



sympy_mobius.py
用参数曲面绘制麦比乌斯带

```
from sympy import *
u, v = symbols("u,v")
```

```
x = (1+v/2*cos(u/2))*cos(u)
y = (1+v/2*cos(u/2))*sin(u)
z = v/2*sin(u/2)

Plot(x, y, z, [u, 0, 2*pi], [v, -1, 1])
```

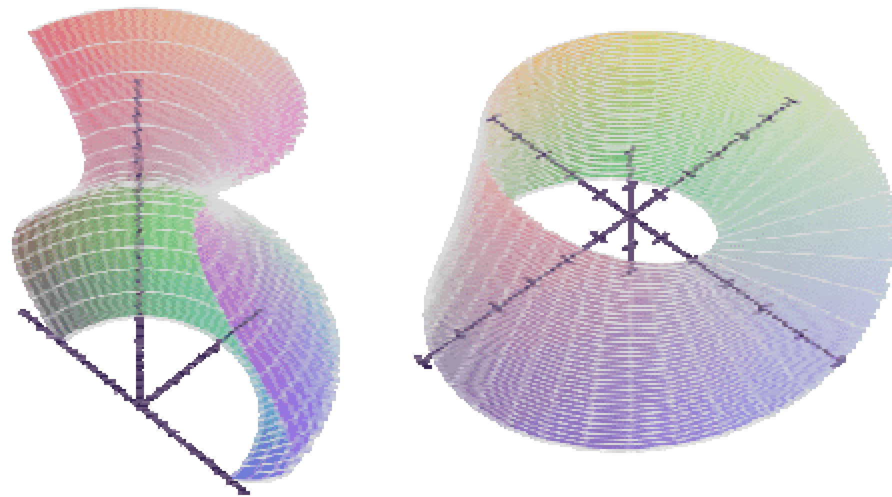


图 4-3　圆柱坐标系中的旋转曲面(左)、用参数曲面绘制麦比乌斯带(右)

控制绘图窗口

在绘图窗口中可以使用快捷键或鼠标对三维场景进行操作:

- 缩放：按住 Page Up 和 Page Down 键，使用鼠标滑轮滚动，或者用鼠标中键进行上下拖动。
- 平移：按住鼠标右键并拖动。
- 旋转：按键 A 和 D、S 和 W、Q 和 E，或者用鼠标左键拖动。
- 默认视图方向：按键 F1 看 XY 平面，按键 F2 看 XZ 平面，按键 F3 看 YZ 平面，按键 F4 看透视图。
- 坐标轴：按键 F5 切换显示，按键 F6 切换颜色。
- 保存图像：按键 F8 将当前场景保存成 PNG 图像。
- 按住 Shift 键可以进行高精度缩放、平移和旋转操作。

第 5 章

matplotlib——绘制精美的图表

matplotlib 是 Python 最著名的绘图库，它提供了一整套和 MATLAB 类似的绘图函数集，十分适合编写短小的脚本程序，进行快速绘图。此外，matplotlib 采用面向对象技术来实现，因此组成图表的各个元素都是对象，在编写较大的应用程序时通过面向对象的方式使用 matplotlib 将更加有效。

matplotlib 的文档十分完备，并且展示页面中有上百幅图表的缩略图及源程序。因此如果读者需要绘制某种类型的图表，只需要在这个页面中“浏览/复制/粘贴”一下，基本上就都能快速解决。



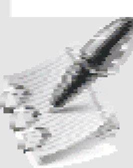
http://matplotlib.sourceforge.net/gallery.html
matplotlib 展示页面的地址

本章在简单介绍 matplotlib 的快速绘图功能之后，将较为深入地分析几个实例，让读者从中学习和理解 matplotlib 绘图的一些基本概念。相信读者在了解本章的内容之后，应该能够根据官方文档和演示程序使用 matplotlib 完美地展示自己的数据。

5.1 快 速 绘 图

5.1.1 使用 pyplot 模块绘图

matplotlib 的 pyplot 模块提供了和 MATLAB 类似的绘图 API，可方便用户快速绘制二维图表。我们先看一个简单的例子：



matplotlib_simple_plot.py
使用 pyplot 模块快速绘图

```
import numpy as np
import matplotlib.pyplot as plt ❶

x = np.linspace(0, 10, 1000)
y = np.sin(x)
```

```
z = np.cos(x**2)

plt.figure(figsize=(8,4)) ❷

plt.plot(x,y,label="$sin(x)$",color="red",linewidth=2) ❸
plt.plot(x,z,"b--",label="$cos(x^2)$") ❹

plt.xlabel("Time(s)") ❺
plt.ylabel("Volt")
plt.title("PyPlot First Example")
plt.ylim(-1.2,1.2)
plt.legend()

plt.show() ❻
```


程序的输出如图 5-1 所示。

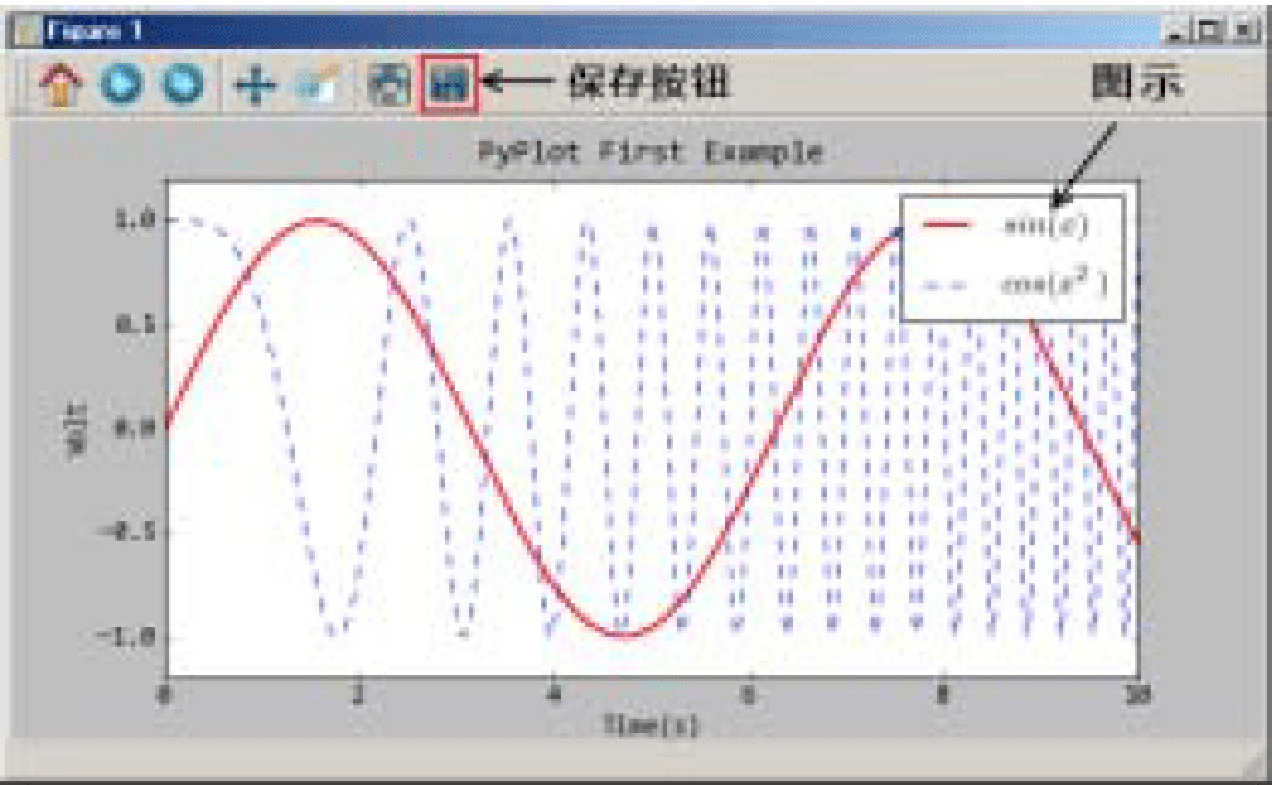


图 5-1　使用 pyplot 模块快速将数据绘制成曲线图

❶首先载入 matplotlib 的绘图模块 pyplot，并重命名为 plt。

pylab 模块

matplotlib 提供了一个名为 pylab 的模块，其中包括了许多 NumPy 和 pyplot 模块中常用的函数，可以方便用户快速进行计算和绘图，十分适合在 IPython 交互式环境中使用。本书使用下面的方式载入 pylab 模块：

```
>>> import pylab as pl
```

❷调用 figure()创建一个 Figure(图表)对象，并且它将成为当前 Figure 对象。也可以不创建 Figure 对象，而是直接调用接下来的 plot()进行绘图，这时 matplotlib 会自动创建一个 Figure 对象。figsize 参数指定 Figure 对象的宽度和高度，单位为英寸。此外还可以使用 dpi 参数指定 Figure 对象的分辨率，即每英寸所表示的像素数，这里使用默认值 80。因此本例中创建的 Figure 对象

的宽度为“8×80 = 640”个像素。但是在显示出绘图窗口之后，用工具栏中的保存按钮将图表保存为图像时，保存的图像的大小是“800×400”像素。这是因为保存图像时会使用不同的 dpi 设置。这个设置保存在 matplotlib 的配置文件中，可以通过如下语句查看它的值：

```
>>> import matplotlib
>>> matplotlib.rcParams["savefig.dpi"]
100
```

因为保存图像时的 dpi 设置为 100，所以保存的图像的宽度是“8×100 = 800”个像素。rcParams 是一个字典，其中保存着从配置文件读入的所有配置，在调用各种绘图函数时，这些配置将会作为各种参数的默认值。后面我们还会对 matplotlib 的配置文件进行详细介绍。

❸创建 Figure 对象之后，接下来调用 plot()在当前的 Figure 对象中绘图。实际上 plot()是在 Axes(子图)对象上绘图，如果当前的 Figure 对象中没有 Axes 对象，那么将会为之创建一个几乎充满整个图表的 Axes 对象，并使此 Axes 对象成为当前的 Axes 对象。plot()的前两个参数是分别表示 X、Y 轴数据的对象，这里使用的是 NumPy 数组。使用关键字参数可以指定所绘制曲线的各种属性：

- label：给曲线指定一个标签名称，此标签将在图示中显示。如果标签字符串的前后有字符'$'，那么 matplotlib 会使用内嵌的 LaTeX 引擎将其显示为数学公式。
- color：指定曲线的颜色，颜色可以用英文单词来表示，也可以用以'#'字符开头的三个 16 进制数来表示，例如'#ff0000'表示红色。还可以使用值在 0 到 1 范围之内的三个元素的元组来表示，例如(1.0, 0.0, 0.0)也表示红色。
- linewidth：指定曲线的宽度，可以不是整数，但可以使用缩写形式的参数名 lw。



使用 LaTex 语法绘制数学公式会极大地降低图表的描绘速度。

❹直接通过第三个参数'b--'指定曲线的颜色和线型，它通过一些易记的符号指定曲线的样式。其中'b'表示蓝色，'--'表示线型为虚线。在 IPython 中输入“plt.plot?”，可以查看格式化字符串及各个参数的详细说明。

❺接下来通过一系列函数设置当前 Axes 对象的各个属性：

- xlabel、ylabel：分别设置 X、Y 轴的标题文字。
- title：设置子图的标题。
- xlim、ylim：分别设置 X、Y 轴的显示范围。
- legend：显示图示，即图中表示每条曲线的标签(label)和样式的矩形区域。

❻最后调用 plt.show()显示出绘图窗口。在通常情况下，show()将会阻塞程序的运行，直到用户关闭绘图窗口。然而在带“-wthread”等参数的 IPython 环境下，show()不会等待窗口关闭。

还可以调用 plt.savefig()将当前的 Figure 对象保存成图像文件，图像格式由图像文件的扩展名决定。下面的程序将当前的图表保存为“test.png”，并且通过 dpi 参数指定图像的分辨率为 120，因此输出图像的宽度为“8×120 = 960”个像素。

```
>>> run matplotlib_simple_plot.py
>>> plt.savefig("test.png", dpi=120)
```



如果关闭了图表窗口，就无法使用 savefig()保存图像。实际上不需要调用 show()显示图表，可以直接用 savefig()将图表保存成图像文件。使用这种方法可以很容易编写出批量输出图表的程序。

savefig()的第一个参数可以是文件名，也可以是和 Python 的文件对象有相同调用接口的对象。例如可以将图像保存到 StringIO 对象中，这样就得到了一个表示图像内容的字符串。这里需要使用 fmt 参数指定保存的图像格式。

```
>>> from StringIO import StringIO
>>> buf = StringIO() # 创建一个用来保存图像内容的 StringIO 对象
>>> plt.savefig(buf, fmt="png") # 将图像以 png 格式保存到 buf 中
>>> buf.getvalue()[:20] # 显示图像内容的前 20 个字节
'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x03 '
```

5.1.2　以面向对象方式绘图

matplotlib 实际上是一套面向对象的绘图库，它所绘制的图表中的每个绘图元素，例如线条、文字、刻度等在内存中都有一个对象与之对应。为了方便快速绘图，matplotlib 通过 pyplot 模块提供了一套和 MATLAB 类似的绘图 API，将众多绘图对象所构成的复杂结构隐藏在这套 API 内部。我们只需要调用 pyplot 模块所提供的函数，就可以实现快速绘图以及设置图表的各种细节。pyplot 模块虽然用法简单，但不适合在较大的应用程序中使用，因此本章将着重介绍如何使用 matplotlib 以面向对象方式编写绘图程序。

为了将面向对象的绘图库封装成只使用函数的调用接口，pyplot 模块的内部保存了当前图表以及当前子图等信息。当前的图表和子图可以使用 gcf()和 gca()获得，它们分别是“Get Current Figure”和“Get Current Axes”首字母的缩写。gcf()获得的是表示图表的 Figure 对象，而 gca()获得的是表示子图的Axes对象。下面我们在IPython中运行上一小节的“matplotlib_simple_plot.py”程序，然后调用 gcf()和 gca()查看当前的 Figure 和 Axes 对象。

```
>>> run matplotlib_simple_plot.py
>>> fig = plt.gcf()
>>> axes = plt.gca()
>>> fig
<matplotlib.figure.Figure object at 0x04B30090>
>>> axes
<matplotlib.axes.AxesSubplot object at 0x04BD8E70>
```

在 pyplot 模块中，许多函数都是对当前的 Figure 或 Axes 对象进行处理，例如前面介绍的

plot()、xlabel()、savefig()等。可以在 IPython 中输入函数名并加“??”，查看这些函数的源代码，了解它们是如何调用各种对象的方法进行绘图处理的。例如下面的例子查看 plot()的源程序，可以看到 plot()实际上会通过 gca()获得当前的 Axes 对象 ax，然后再调用它的 plot()方法实现真正的绘图。请读者使用类似的方法，查看 pyplot 模块的其他函数是如何对各种绘图对象进行封装的。

```
>>> plt.plot??
...
def plot(*args, **kwargs):
    ax = gca()
    ...
    try:
        ret = ax.plot(*args, **kwargs)
        ...
    finally:
        ax.hold(washold)
```

5.1.3 配置属性

使用 matplotlib 绘制的图表的每个组成部分都和一个对象对应，可以通过调用这些对象的属性设置方法 set_*()或 pyplot 模块的属性设置函数 setp()来设置它们的属性值。例如，plot()返回一个元素类型为 Line2D 的列表，下面的例子设置 Line2D 对象的属性：

```
>>> x = np.arange(0, 5, 0.1)
>>> line = plt.plot(x, x*x)[0] # plot 返回一个列表
>>> line.set_antialiased(False) # 调用 Line2D 对象的 set_*()方法来设置属性值
```

在上面的例子中，通过调用 Line2D 对象的 set_antialiased(False)，关闭了它在图表中对应曲线的反锯齿效果。下面的语句同时绘制正弦和余弦两条曲线，lines 是一个有两个 Line2D 对象的列表：

```
>>> lines = plt.plot(x, np.sin(x), x, np.cos(x))
```

调用 setp()可以同时配置多个对象的属性，这里同时设置两条曲线的颜色和线宽：

```
>>> plt.setp(lines, color="r", linewidth=2.0)
```

同样，可以通过调用 Line2D 对象的 get_*()或 plt.getp()来获取对象的属性值：

```
>>> line.get_linewidth()
1.0
>>> plt.getp(lines[0], "color") # 返回 color 属性
'r'
```

```
>>> plt.getp(lines[1]) # 输出全部属性
alpha = 1.0
animated = False
antialiased or aa = True
axes = Axes(0.125,0.1;0.775x0.8)
...
```

注意，getp()和 setp()不同，它只能对一个对象进行操作，它有两种用法：

- 指定属性名：返回对象中某个属性的值。
- 不指定属性名：输出对象中所有属性和值。

下面通过 getp()查看 Figure 对象的属性：

```
>>> f = plt.gcf()
>>> plt.getp(f)
alpha = 1.0
animated = False
...
```

Figure 对象的 axes 属性是一个列表，它保存图表中的所有子图对象。下面的程序查看当前图表的 axes 属性，可以看出其中包含 gca()所获得的当前子图对象：

```
>>> plt.getp(f, "axes")
[<matplotlib.axes.AxesSubplot object at 0x05CDD170>]
>>> plt.gca()
<matplotlib.axes.AxesSubplot object at 0x05CDD170>
```

用 plt.getp()可以继续获取 AxesSubplot 对象的属性，例如它的 lines 属性为子图中的 Line2D 对象列表：

```
>>> alllines = plt.getp(plt.gca(), "lines")
>>> alllines
<a list of 3 Line2D objects>
>>> alllines[0] == line # 其中的第一条曲线就是最开始绘制的那条曲线
True
```

通过这种方法可以很容易查看对象的属性值以及各个对象之间的关系，找到需要配置的属性。

由于 matplotlib 实际上是一套面向对象的绘图库，因此也可以直接获取对象的属性，例如：

```
>>> f.axes
[<matplotlib.axes.AxesSubplot object at 0x05CDD170>]
>>> f.axes[0].lines
<a list of 3 Line2D objects>
```

5.1.4　绘制多个子图

一个 Figure 对象可以包含多个子图(Axes)，在 matplotlib 中用 Axes 对象表示一个绘图区域，在本书中称之为子图。在前面的例子中，Figure 对象只包括一个子图。可以使用 subplot()快速绘制包含多个子图的图表，它的调用形式如下：

```
subplot(numRows, numCols, plotNum)
```

图表的整个绘图区域被等分为 numRows 行和 numCols 列，然后按照从左到右、从上到下的顺序对每个区域进行编号，左上区域的编号为 1。plotNum 参数指定创建的 Axes 对象所在的区域。如果 numRows、numCols 和 plotNum 三个参数都小于 10，就可以把它们缩写成一个整数，例如 subplot(323)和 subplot(3,2,3)的含义相同。如果新创建的子图和之前创建的子图区域有重叠的部分，之前的子图将被删除。

下面的程序创建如图 5-2 所示的 3 行 2 列共 6 个子图，并通过 axisbg 参数给每个子图设置不同的背景颜色(见文前彩插)。

```
for idx, color in enumerate("rgbyck"):
    plt.subplot(321+idx, axisbg=color)
plt.show()
```

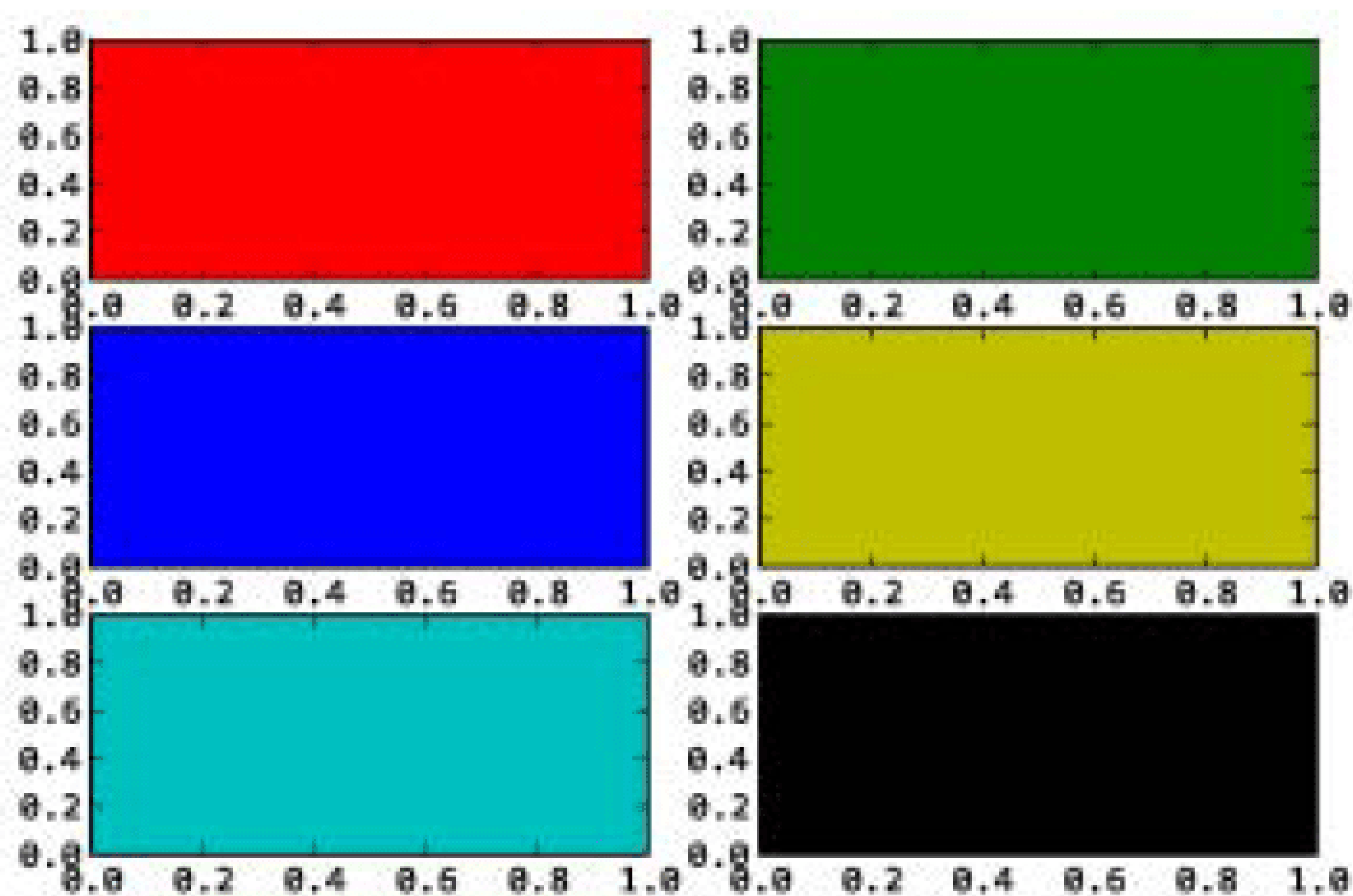


图 5-2　用 subplot()在当前的 Figure 对象中创建 6 个子图

如果希望某个子图占据整行或整列，可以按如下形式调用 subplot()：

```
plt.subplot(221) # 第一行的左图
plt.subplot(222) # 第一行的右图
plt.subplot(212) # 第二整行
plt.show()
```

程序的输出如图 5-3 所示。

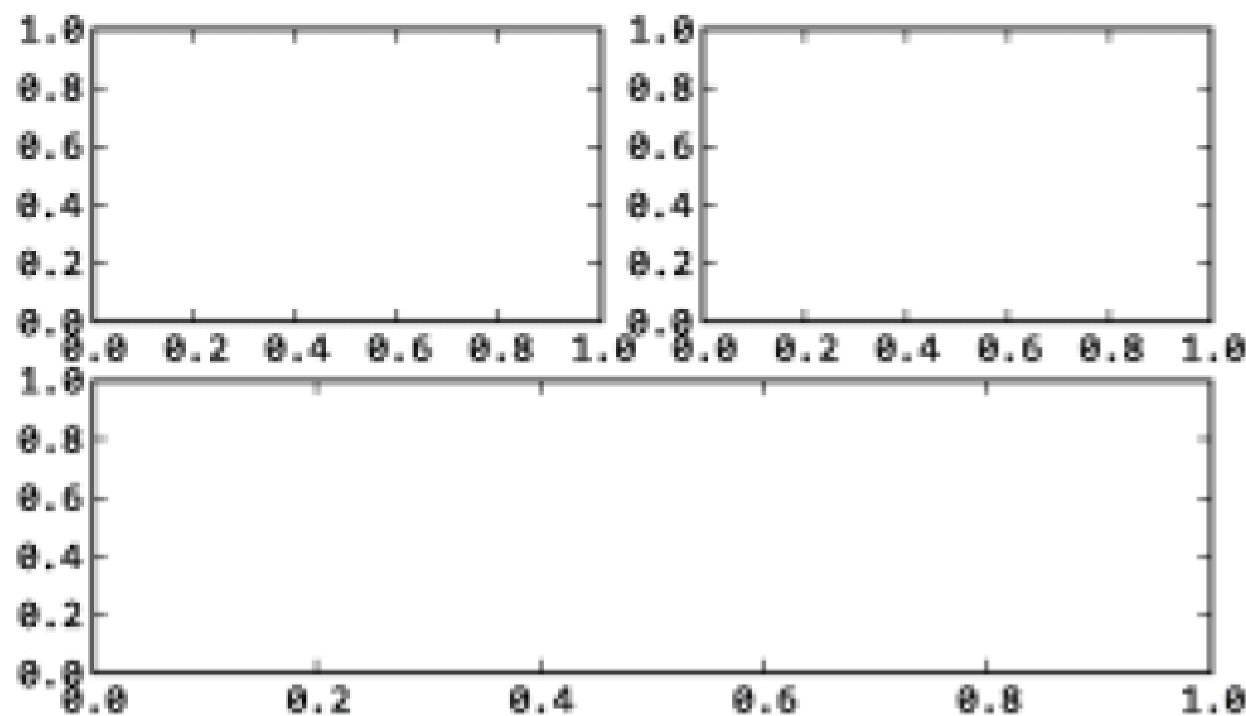


图 5-3　将 Figure 分为三个子图

在绘图窗口的工具栏中，有一个名为“Configure Subplots”的按钮，单击它将弹出用于调节子图间距和子图与图表边框距离的对话框。也可以在程序中调用 subplots_adjust()调节这些参数，它有 left、right、bottom、top、wspace 和 hspace 共 6 个参数，这些参数与对话框中的各个控件对应。参数的取值范围为 0 到 1，它们是对图表绘图区域的宽和高进行正规化之后的坐标或长度。

subplot()返回它所创建的 Axes 对象，可以将它用变量保存起来，然后用 sca()交替让它们成为当前 Axes 对象，并调用 plot()在其中绘图。如果需要同时绘制多幅图表，可以给 figure()传递一个整数参数指定 Figure 对象的序号，如果序号所指定的 Figure 对象已经存在，将不创建新的对象，而只是让它成为当前的 Figure 对象。下面的程序演示了如何依次在不同图表的不同子图中绘制曲线。



matplotlib_multi_figure.py
同时在多幅图表、多个子图中进行绘图

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1) # 创建图表 1
plt.figure(2) # 创建图表 2
ax1 = plt.subplot(211) # 在图表 2 中创建子图 1
ax2 = plt.subplot(212) # 在图表 2 中创建子图 2

x = np.linspace(0, 3, 100)
for i in xrange(5):
    plt.figure(1)  ❶ # 选择图表 1
    plt.plot(x, np.exp(i*x/3))
```

```
    plt.sca(ax1)   ❷ # 选择图表 2 的子图 1
    plt.plot(x, np.sin(i*x))
    plt.sca(ax2)  # 选择图表 2 的子图 2
    plt.plot(x, np.cos(i*x))

plt.show()
```

首先通过 figure()创建了两个图表，它们的序号分别为 1 和 2。然后在图表 2 中创建了上下并排的两个子图，并用变量 ax1 和 ax2 保存。

在循环中，❶先调用 figure(1)让图表 1 成为当前图表，并在其中绘图。❷然后调用 sca(ax1)和 sca(ax2)分别让子图 ax1 和 ax2 成为当前子图，并在其中绘图。当它们成为当前子图时，包含它们的图表 2 也自动成为当前图表，因此不需要调用 figure(2)。依次在图表 1 和图表 2 的两个子图之间切换，逐步在其中添加新的曲线，效果如图 5-4 所示(见文前彩插)。

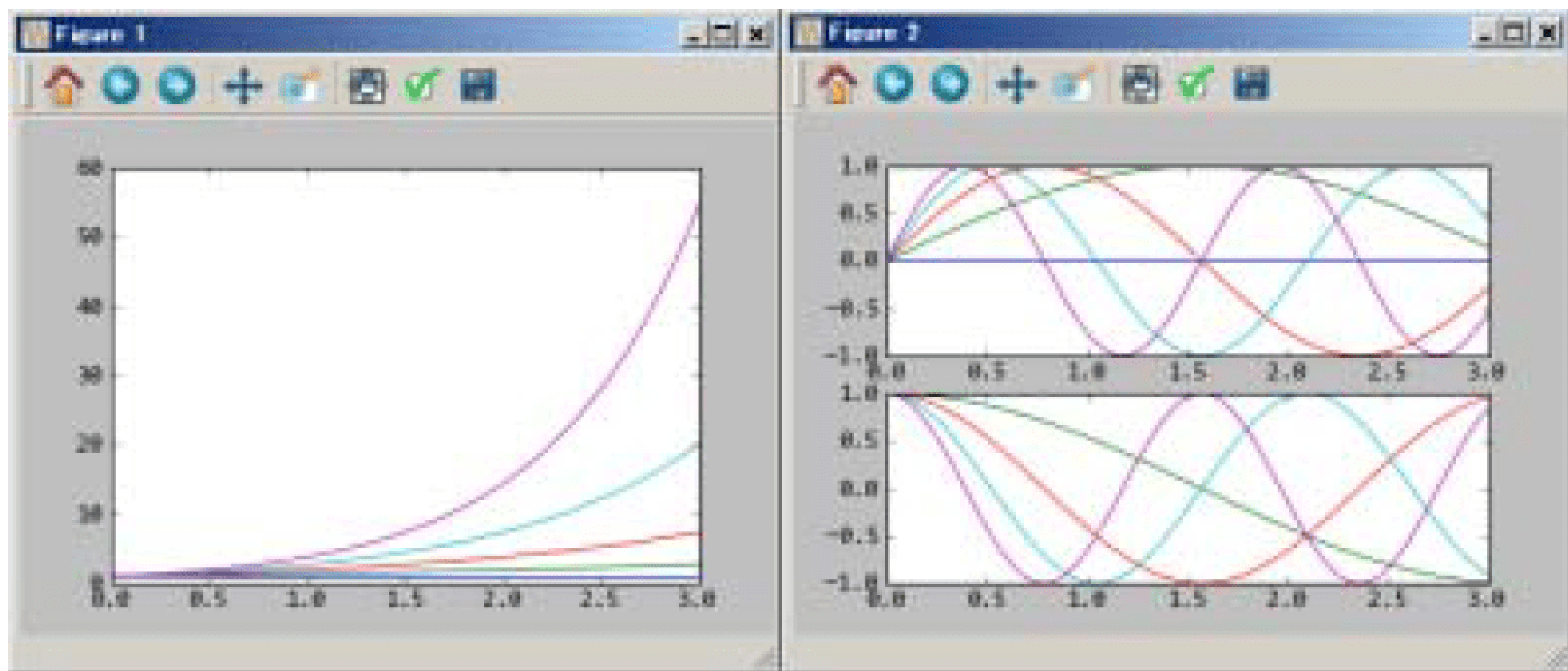


图 5-4　同时在多幅图表、多个子图中进行绘图

5.1.5　配置文件

绘制一幅图表需要对许多对象的属性进行配置，例如颜色、字体、线型，等等。我们在绘图时，并没有逐一对这些属性进行配置，许多都直接采用了 matplotlib 的默认配置。matplotlib 将这些默认配置保存在一个名为“matplotlibrc”的配置文件中，通过修改配置文件，我们可以修改图表的默认样式。

在 matplotlib 中可以使用多个“matplotlibrc”配置文件，它们的搜索顺序如下(顺序靠前的配置文件将会被优先采用)：

- 当前路径：程序的当前路径。
- 用户配置路径：通常在用户文件夹的“.matplotlib”目录下，可以通过环境变量 MATPLOTLIBRC 修改它的位置。
- 系统配置路径：保存在 matplotlib 安装目录下的 mpl-data 中。

通过下面的语句可以获取用户配置路径：

```
>>> import matplotlib
>>> matplotlib.get_configdir()
'C:\\Documents and Settings\\用户名\\.matplotlib'
```

通过下面的语句可以获得目前使用的配置文件的路径：

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'C:\\Python26\\lib\\site-packages\\matplotlib\\mpl-data\\matplotlibrc'
```

由于在当前路径和用户配置路径中都没有找到配置文件，因此最后使用的是系统配置路径下的配置文件。如果读者将 matplotlibrc 复制一份到脚本的当前目录(例如 c:\zhang\doc)下：

```
>>> import os
>>> os.getcwd()
'C:\\zhang\\doc'
```

复制配置文件之后再查看配置文件的路径，就会发现它变成了当前目录下的配置文件：

```
>>> matplotlib.matplotlib_fname()
'C:\\zhang\\doc\\matplotlibrc'
```

如果读者使用文本编辑器打开此配置文件，就会发现它实际上是一个字典。为了对众多的配置进行区分，字典的键根据配置的种类用“.”分为多段。

配置文件的读入可以使用 rc_params()，它返回一个配置字典：

```
>>> matplotlib.rc_params()
{'agg.path.chunksize': 0,
 'axes.axisbelow': False,
 'axes.edgecolor': 'k',
 'axes.facecolor': 'w',
 ... ...
```

matplotlib 模块在载入时会调用 rc_params()，并把得到的配置字典保存到 rcParams 变量中：

```
>>> matplotlib.rcParams
{'agg.path.chunksize': 0,
'axes.axisbelow': False,
... ...
```

matplotlib 将使用 rcParams 字典中的配置进行绘图。用户可以直接修改此字典中的配置，所做的改动会反映到此后创建的绘图元素上。例如使用下面代码绘制的折线将带有圆形的点标识符：

```
>>> matplotlib.rcParams["lines.marker"] = "o"
>>> plt.plot([1,2,3,2])
>>> plt.show()
```

为了方便对配置字典进行设置，可以使用 rc()。下面的例子可同时配置点标识符、线宽和颜色：

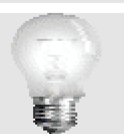
```
>>> matplotlib.rc("lines", marker="x", linewidth=2, color="red")
```

如果希望恢复到默认的配置(matplotlib 载入时从配置文件读入的配置)，可以调用 rcdefaults()：

```
>>> matplotlib.rcdefaults()
```

如果手工修改了配置文件，希望重新从配置文件载入最新的配置，可以调用：

```
>>> matplotlib.rcParams.update( matplotlib.rc_params() )
```



通过 pyplot 模块也可以使用 rcParams、rc 和 rcdefaults。

5.1.6 在图表中显示中文

matplotlib 默认配置文件中使用的字体无法正确显示中文。为了让图表能正确显示中文，可以有如下几种解决方案：

- 在程序中直接指定字体。
- 在程序开头修改配置字典 rcParams。
- 修改配置文件。

在 matplotlib 中可以通过字体名指定字体，而每个字体名都与一个字体文件相对应。通过下面的程序可以获得所有可用的字体列表：

```
>>> from matplotlib.font_manager import fontManager
>>> fontManager.ttflist
[<Font 'cmex10' (cmex10.ttf) normal normal 400 normal>,
 <Font 'Bitstream Vera Sans Mono' (VeraMoBd.ttf) normal normal 700 normal>,
 ...
]
```

fontManager.ttflist 是 matplotlib 的系统字体索引列表，其中的每个元素都是表示字体的 Font 对象。例如，由第一个 Font 对象可知，字体名“cmex10”与字体文件“cmex10.ttf”相对应。使用下面的语句可获得字体文件的全路径和字体名：

```
>>> fontManager.ttflist[0].name
```

```
'cmex10'
>>> fontManager.ttflist[0].fname
'C:\\Python26\\lib\\site-packages\\matplotlib\\mpl-data\\fonts\\ttf\\cmex10.ttf'
```

由字体文件的路径可知，“cmex10”是 matplotlib 自带的字体。下面的程序利用字体索引列表中的字体显示中文文字，效果如图 5-5 所示。

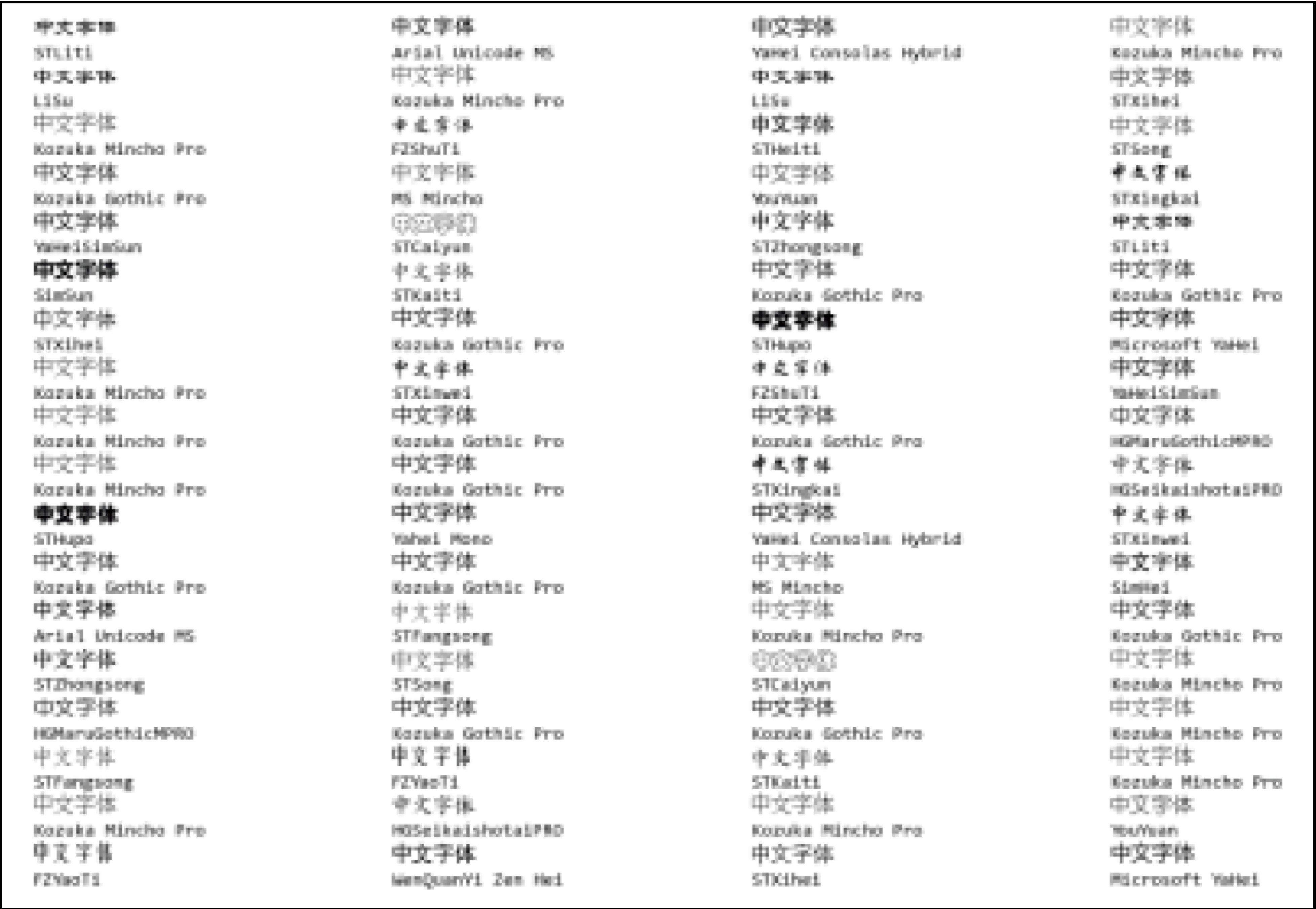


图 5-5　显示系统中的中文字体

matplotlib_fonts.py
显示所有的中文字体

```
from matplotlib.font_manager import fontManager
import matplotlib.pyplot as plt
import os

fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(111)
plt.subplots_adjust(0, 0, 1, 1, 0, 0)
plt.xticks([])
plt.yticks([])
x, y = 0.05, 0.08
```

```
fonts = [font.name for font in fontManager.ttflist if
    os.path.exists(font.fname) and os.stat(font.fname).st_size>1e6] ❶
font = set(fonts)
dy = (1.0-y)/(len(fonts)/4 + (len(fonts)%4!=0))
for font in fonts:
    t = ax.text(x, y, u"中文字体", {'fontname':font, 'fontsize':14}, transform=ax.transAxes) ❷
    ax.text(x, y-dy/2, font, transform=ax.transAxes)
    x += 0.25
    if x >= 1.0:
        y += dy
        x = 0.05
plt.show()
```

❶利用 os 模块中的 stat()获取字体文件的大小，并保留字体索引列表中大于 1MB 的所有字体文件。由于中文字体文件通常都很大，因此使用这种方法可以粗略地找出所有的中文字体文件。

❷调用子图对象的 text()在其中添加文字，注意文字必须是 Unicode 字符串。通过一个描述字体的字典指定文字的字体：'fontname'键对应的值就是字体名。

由于 matplotlib 只搜索 TTF 字体文件，因此无法通过上述方法使用 Windows 中 Fonts 目录下的许多复合字体文件(*.ttc)。可以直接创建使用字体文件的 FontProperties 对象，并使用此对象指定图表中各种文字的字体。下面是一个例子：



matplotlib_simsun_font.py
使用 TTC 字体文件

```
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
import numpy as np
font = FontProperties(fname=r"c:\windows\fonts\simsun.ttc", size=14) ❶
t = np.linspace(0, 10, 1000)
y = np.sin(t)
plt.plot(t, y)
plt.xlabel(u"时间", fontproperties=font) ❷
plt.ylabel(u"振幅", fontproperties=font)
plt.title(u"正弦波", fontproperties=font)
plt.show()
```

❶创建一个描述字体属性的 FontProperties 对象，并设置其 fname 属性为字体文件的绝对路径。❷通过 fontproperties 参数将 FontProperties 对象传递给显示文字的函数。

还可以通过字体工具将 TTC 字体文件分解为多个 TTF 字体文件，并将其复制到系统的字体文件夹中。为了缩短启动时间，matplotlib 不会每次启动时都重新扫描所有的字体文件并创建

字体索引列表，因此在复制完字体文件之后，需要运行下面的语句以重新创建字体索引列表：

```
>>> from matplotlib.font_manager import _rebuild
>>> _rebuild()
```

还可以直接修改配置字典，设置默认字体，这样就不需要在每次绘制文字时设置字体了。例如：

```
>>> plt.rcParams["font.family"] = "SimHei"
>>> plt.plot([1,2,3])
>>> plt.xlabel(0.5,0.5,u"中文字体")
```

或者修改前面介绍的配置文件，修改其中的“font.family”配置为：

```
font.family: SimHei
```

注意“SimHei”是字体名，请读者运行“matplotlib_fonts.py”查看系统中所有可用的中文字体名。

5.2 Artist 对象

matplotlib 是一套面向对象的绘图库，它有三个层次：

- backend_bases.FigureCanvas：图表的绘制领域。
- backend_bases.Renderer：知道如何在 FigureCanvas 对象上绘图。
- artist.Artist：知道如何使用 Renderer 在 FigureCanvas 对象上绘图。

FigureCanvas 和 Renderer 需要处理底层的绘图操作，例如在 wxPython 界面上绘图或者使用 PostScript 在 PDF 文件中绘图。Artist 对象则处理所有的高层结构，例如处理图表、文字和曲线等各种绘图元素的绘制和布局。通常我们只和 Artist 对象打交道，而不需要关心底层是如何实现绘图细节的。

Artist 对象分为简单类型和容器类型两种。简单类型的 Artist 对象是标准的绘图元件，例如 Line2D、Rectangle、Text、AxesImage 等。而容器类型则可以包含许多简单类型的 Artist 对象，使它们组织成一个整体，例如 Axis、Axes、Figure 等。

直接创建 Artist 对象进行绘图的流程如下：

(1) 创建 Figure 对象。

(2) 为 Figure 对象创建一个或多个 Axes 对象。

(3) 调用 Axes 对象的方法创建各种简单类型的 Artist 对象。

在下面的程序中，首先调用 figure()创建 Figure 对象。figure()是一个辅助函数，可以帮助我们创建 Figure 对象，它会进行许多初始化操作，因此不建议直接使用 figure()创建。然后调用 Figure 对象的 add_axes()在其中创建一个 Axes 对象，add_axes()的参数是一个形如[left, bottom,

width, height]的列表，这些数值分别指定所创建的 Axes 对象在 Figure 对象中的位置和大小，各个值的取值范围都在 0 到 1 之间：

```
>>> fig = plt.figure()
>>> ax = fig.add_axes([0.15, 0.1, 0.7, 0.3])
```

然后调用 Axes 对象的 plot()进行绘图，plot()可以绘制一条曲线，并且返回表示此曲线的 Line2D 对象：

```
>>> line = ax.plot([1,2,3],[1,2,1])[0] # 返回的是一个只含有一个元素的列表
>>> ax.lines
[<matplotlib.lines.Line2D object at 0x0637A3D0>]
>>> line
<matplotlib.lines.Line2D object at 0x0637A3D0>
```

Axes 对象的 lines 属性是一个包含其所有曲线的列表，如果继续运行 ax.plot()，所创建的 Line2D 对象就都会添加到此列表中。如果想删除某条曲线，直接从此列表中删除它即可。

Axes 对象还包括许多其他的 Artists 对象，例如我们可以通过 set_xlabel()设置其 X 轴上的标题：

```
>>> ax.set_xlabel("time")
```

如果查看 set_xlabel()的源代码，就会发现它是通过下面的语句实现的：

```
self.xaxis.set_label_text(xlabel)
```

如果一直跟踪下去，会发现 Axes 对象的 xaxis 属性是一个 XAxis 对象：

```
>>> ax.xaxis
<matplotlib.axis.XAxis object at 0x06343230>
```

XAxis 对象的 label 属性是一个 Text 对象：

```
>>> ax.xaxis.label
<matplotlib.text.Text object at 0x06343290>
```

而 Text 对象的_text 属性为我们设置的值：

```
>>> ax.xaxis.label._text
'time'
```

这些都是 Artist 对象，因此也可以调用它们的 get_*()来获得相应的属性值：

```
>>> ax.xaxis.label.get_text()
'time'
```

5.2.1 Artist 对象的属性

通过前面的介绍我们已经知道，图表中的每个绘图元素都用一个 Artist 对象来表示，而每个 Artist 对象都有一大堆属性用来控制其显示效果。例如 Figure 对象和 Axes 对象都有 patch 属性作为其背景，它是一个 Rectangle 对象。通过设置它的属性可以修改图表的背景颜色或透明度等属性，下面的例子将图表的背景颜色设置为绿色：

```
>>> fig = plt.figure()
>>> fig.show()
>>> fig.patch.set_color("g") # 设置背景颜色为绿色
>>> fig.canvas.draw() # 重绘界面
```

注意：在调用 set_color()设置好背景颜色之后，设置的背景颜色并不会立即在界面上显示出来，还需要调用 fig.canvas.draw()才能更新界面显示。

下面是所有 Artist 对象都拥有的一些属性：

- alpha：透明度，值在 0 到 1 之间，0 为完全透明，1 为完全不透明。
- animated：布尔值，在绘制动画效果时使用。
- axes：拥有此 Artist 对象的 Axes 对象，可能为 None。
- clip_box：对象的裁剪框。
- clip_on：是否裁剪。
- clip_path：裁剪的路径。
- contains：判断指定点是否在对象之上的函数。
- figure：拥有此 Artist 对象的 Figure 对象，可能为 None。
- label：文本标签。
- picker：控制 Artist 对象的选取。
- transform：控制偏移、旋转、缩放等坐标变换。
- visible：是否可见。
- zorder：控制绘图顺序。

Artist 对象的所有属性都可以通过相应的 get_*()和 set_*()方法进行读写，例如下面的语句将新绘制的曲线对象的 alpha 属性设置为 0.5，从而使它变成半透明：

```
>>> line = plt.plot([1,2,3,2,1], lw=4)[0]
>>> line.set_alpha(0.5)
```

可以使用 set()一次设置多个属性：

```
>>> line.set(alpha=0.5, zorder=2)
```

使用前面介绍的 getp()可以方便地输出 Artist 对象的所有属性名及其对应值：

```
>>> plt.getp(fig.patch)
```

```
    aa = True
    alpha = 1.0
    animated = False
    antialiased or aa = True
    ... ...
```

5.2.2 Figure 容器

现在我们已经知道如何观察和修改某个已知的 Artist 对象的属性，接下来要解决如何找到指定的 Artist 对象。前面介绍过 Artist 对象分为容器类型和简单类型两种，这一节让我们详细看看容器类型。

在构成图表的各种 Artist 对象中，最上层的 Artist 对象是 Figure，它包含组成图表的所有元素。当调用其 add_subplot()或 add_axes()方法向图表中添加子图时，这些子图都将添加到 axes 属性列表中，同时这两个方法将返回新创建的 Axes 对象。注意：add_subplot()和 add_axes()所返回的对象的类型有所不同，分别为 AxesSubplot 和 Axes，AxesSubplot 是 Axes 的派生类。

```
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(211)
>>> ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
>>> ax1
<matplotlib.axes.AxesSubplot object at 0x056BCA90>
>>> ax2
<matplotlib.axes.Axes object at 0x056BC910>
>>> fig.axes
[<matplotlib.axes.AxesSubplot object at 0x056BCA90>,
<matplotlib.axes.Axes object at 0x056BC910>]
```

为了支持 gca()等函数，Figure 对象的内部保存有当前子图的信息，因此不建议直接对 axes 属性进行列表操作，而应该使用 add_subplot()、add_axes()、delaxes()等方法进行子图的添加和删除操作。但是使用 for 循环对 axes 属性中的每个元素进行操作是没有问题的，下面的语句将打开所有子图的栅格显示：

```
>>> for ax in fig.axes: ax.grid(True)
```

Figure 对象可以拥有自己的文字、线条以及图像等简单类型的 Artist 对象。Artist 对象的默认坐标系统以像素点为单位，但是可以通过设置其 transform 属性修改所使用的坐标系。例如 Figure 对象的坐标系是以图表的左下角为坐标原点(0,0)，右上角的坐标为(1,1)。关于坐标变换在后面的章节还会进行详细介绍。下面的程序创建一个 Figure 对象，并在其中添加两条直线，效果如图 5-6 所示：

```
>>> from matplotlib.lines import Line2D
```

```
>>> fig = plt.figure()
>>> line1 = Line2D([0,1],[0,1], transform=fig.transFigure, figure=fig, color="r")
>>> line2 = Line2D([0,1],[1,0], transform=fig.transFigure, figure=fig, color="g")
>>> fig.lines.extend([line1, line2])
>>> fig.show()
```

为了让创建的 Line2D 对象使用 Figure 对象的坐标系，我们将 Figure 对象的 transFigure 属性赋给 Line2D 对象的 transform 属性。为了让 Line2D 对象知道它是在 Figure 对象中，我们还设置其 figure 属性为 fig。最后还需要将这两个 Line2D 对象添加到 Figure 对象的 lines 属性列表中。

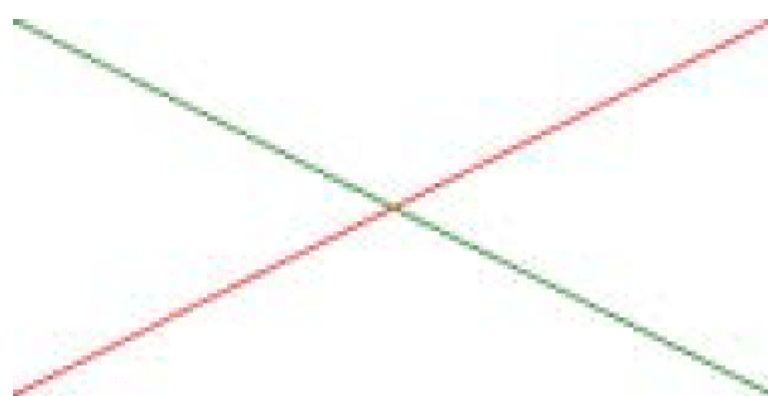


图 5-6　在 Figure 对象中手工绘制直线

下面列出了 Figure 对象中包含其他 Artist 对象的属性：

- axes：Axes 对象列表。
- patch：作为背景的 Rectangle 对象。
- images：FigureImage 对象列表，用于显示图像。
- legends：Legend 对象列表，用于显示图示。
- lines：Line2D 对象列表。
- patches：Patch 对象列表。
- texts：Text 对象列表，用于显示文字。

5.2.3　Axes 容器

Axes 容器(子图)是整个 matplotlib 的核心，它包含了组成图表的众多 Artist 对象，并且有许多方法、函数，用于帮助我们创建和修改这些对象。和 Figure 容器一样，它有一个 patch 属性作为背景，当它是笛卡尔坐标时，patch 属性是一个 Rectangle 对象；而当它是极坐标时，patch 属性则是 Circle 对象。例如下面的语句将 Axes 对象的背景颜色设置为绿色：

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.patch.set_facecolor("green")
```

当调用 Axes 对象的绘图方法 plot()时，它将创建一组 Line2D 对象，并将它们添加进 Axes 对象的 lines 属性中，最后返回包含所有创建的 Line2D 对象的列表。plot()的所有关键字参数都将传递给这些 Line2D 对象以设置它们的属性：

```
>>> x, y = np.random.rand(2, 100)
>>> line = ax.plot(x, y, "-", color="blue", linewidth=2)[0]
>>> line
<matplotlib.lines.Line2D object at 0x03007030>
```

```
>>> ax.lines
[<matplotlib.lines.Line2D object at 0x03007030>]
```

注意：plot()返回的是一个 Line2D 对象列表，因为我们可以传递多组 X-Y 轴的数据给 plot()，同时绘制多条曲线。

与 plot()类似，绘制柱状图的函数 bar()和绘制直方统计图的函数 hist()将创建一个 Patch 对象列表，每个元素实际上都是从 Patch 类派生的 Rectangle 对象，所创建的 Patch 对象都添加进 Axes 对象的 patches 属性中：

```
>>> ax = fig.add_subplot(111)
>>> n, bins, rects = ax.hist(np.random.randn(1000), 50, facecolor="blue")
>>> rects
<a list of 50 Patch objects>
>>> rects[0]
<matplotlib.patches.Rectangle object at 0x05BC2350>
>>> ax.patches[0]
<matplotlib.patches.Rectangle object at 0x05BC2350>
```

一般我们不会直接对 lines 或 patches 属性进行操作，而是调用 add_line()或 add_patch()等方法，这些方法可以帮助我们完成许多属性的设置工作。下面看一个例子，首先创建一个 Axes 对象 ax 和一个 Rectangle 对象 rect：

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> rect = plt.Rectangle((1,1), width=5, height=12)
```

查看 rect 的 axes 和 transform 属性：

```
>>> print rect.get_axes() # rect 的 axes 属性为空
None
>>> rect.get_transform() # rect 的 transform 属性为默认值
BboxTransformTo(Bbox(array([[  1.,   1.],
       [  6.,  13.]])))
```

然后通过 add_patch()将 rect 添加到 ax 中，再次查看 rect 的 axes 和 tranform 属性：

```
>>> ax.add_patch(rect) # 将 rect 添加到 ax 中
<matplotlib.patches.Rectangle object at 0x05C34E50>
>>> rect.get_axes() # 于是 rect 的 axes 属性就是 ax
<matplotlib.axes.AxesSubplot object at 0x05C09CB0>
>>> rect.get_transform()     #和 ax.transData 相同
... # 太长，省略
>>> ax.transData
... # 太长，省略
```

由上面的例子可以看出，add_patch()帮助我们设置了 rect 对象的 axes 和 transform 属性。接下来，为了完整显示 rect，调用 ax 的 autoscale_view()方法让它自动调节 X-Y 轴的显示范围：

```
>>> ax.get_xlim() # ax 的 X 轴范围为 0 到 1，无法显示完整的 rect
(0.0, 1.0)
>>> ax.dataLim._get_bounds() # 数据的范围和 rect 的大小一致
(1.0, 1.0, 5.0, 12.0)
>>> ax.autoscale_view() # 自动调整坐标轴范围
>>> ax.get_xlim() # 于是 X 轴可以完整显示 rect
(1.0, 6.0)
>>> plt.show()
```

下面列出了 Axes 对象中可以包含其他 Artist 对象的属性：

- artists：Artist 对象列表。
- patch：作为 Axes 背景的 Patch 对象，可以是 Rectangle 或 Circle。
- collections：Collection 对象列表。
- images：AxesImage 对象列表。
- legends：Legend 对象列表。
- lines：Line2D 对象列表。
- patches：Patch 对象列表。
- texts：Text 对象列表。
- xaxis：XAxis 对象。
- yaxis：YAxis 对象。

表 5-1 列出了 Axes 对象中各种创建其他 Artist 对象的方法：

表 5-1　Axes 对象中创建其他 Artist 对象的方法

Axes 的方法	所创建的对象	添加进的列表
annotate	Annotate	texts
bars	Rectangle	patches
errorbar	Line2D, Rectangle	lines,patches
fill	Polygon	patches
hist	Rectangle	patches
imshow	AxesImage	images
legend	Legend	legends
plot	Line2D	lines
scatter	PolygonCollection	Collections
text	Text	texts

我们以绘制散列图(scatter)为例，演示绘图过程中所创建的各个对象：

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> t = ax.scatter(np.random.rand(20), np.random.rand(20))
>>> t # 返回值为CircleCollection对象
<matplotlib.collections.CircleCollection object at 0x06004230>
>>> ax.collections # 返回的对象已经添加到了 collections 列表中
[<matplotlib.collections.CircleCollection object at 0x06004230>]
>>> fig.show()
>>> t.get_sizes() # 获得Collection的点数
20
```

5.2.4 Axis 容器

Axis 容器包括坐标轴上的刻度线、刻度标签、坐标网格以及坐标轴标题等内容。刻度包括主刻度和副刻度，分别通过 get_major_ticks()和 get_minor_ticks()方法获得。每个刻度线都是一个 XTick 或 YTick 对象，它包括实际的刻度线和刻度标签。为了方便访问刻度线和文本，Axis 对象提供了 get_ticklabels()和 get_ticklines()方法，可以直接获得刻度标签和刻度线。

下面我们看一个例子，首先进行绘图并得到当前子图的 X 轴对象 axis：

```
>>> plt.plot([1,2,3],[4,5,6])
[<matplotlib.lines.Line2D object at 0x0AD3B670>]
>>> plt.show()
>>> axis = plt.gca().xaxis
```

获得 axis 对象的刻度位置列表：

```
>>> axis.get_ticklocs()
array([ 1. ,  1.5,  2. ,  2.5,  3. ])
```

下面获得 axis 对象的刻度标签以及标签中的文字：

```
>>> axis.get_ticklabels() # 获得刻度标签列表
<a list of 5 Text major ticklabel objects>
>>> [x.get_text() for x in axis.get_ticklabels()] # 获得刻度的文本字符串
[u'1.0', u'1.5', u'2.0', u'2.5', u'3.0']
```

下面获得 X 轴上表示主刻度线的列表，我们看到 X 轴上共有 10 条刻度线，它是图 5-7 中上下两个 X 轴上的所有刻度线：

```
>>> axis.get_ticklines()
<a list of 10 Line2D ticklines objects>
```

而由于图 5-7 中没有副刻度线，因此副刻度线列表的长度为 0：

```
>>> axis.get_ticklines(minor=True) # 获得副刻度线列表
<a list of 0 Line2D ticklines objects>
```

获得刻度线或刻度标签之后，可以设置其各种属性。下面设置刻度线为绿色粗线，文本为红色并且旋转 45°，最终的结果如图 5-7 所示：

```
>>> for label in axis.get_ticklabels():
...     label.set_color("red")
...     label.set_rotation(45)
...     label.set_fontsize(16)
...
>>> for line in axis.get_ticklines():
...     line.set_color("green")
...     line.set_markersize(25)
...     line.set_markeredgewidth(3)
```

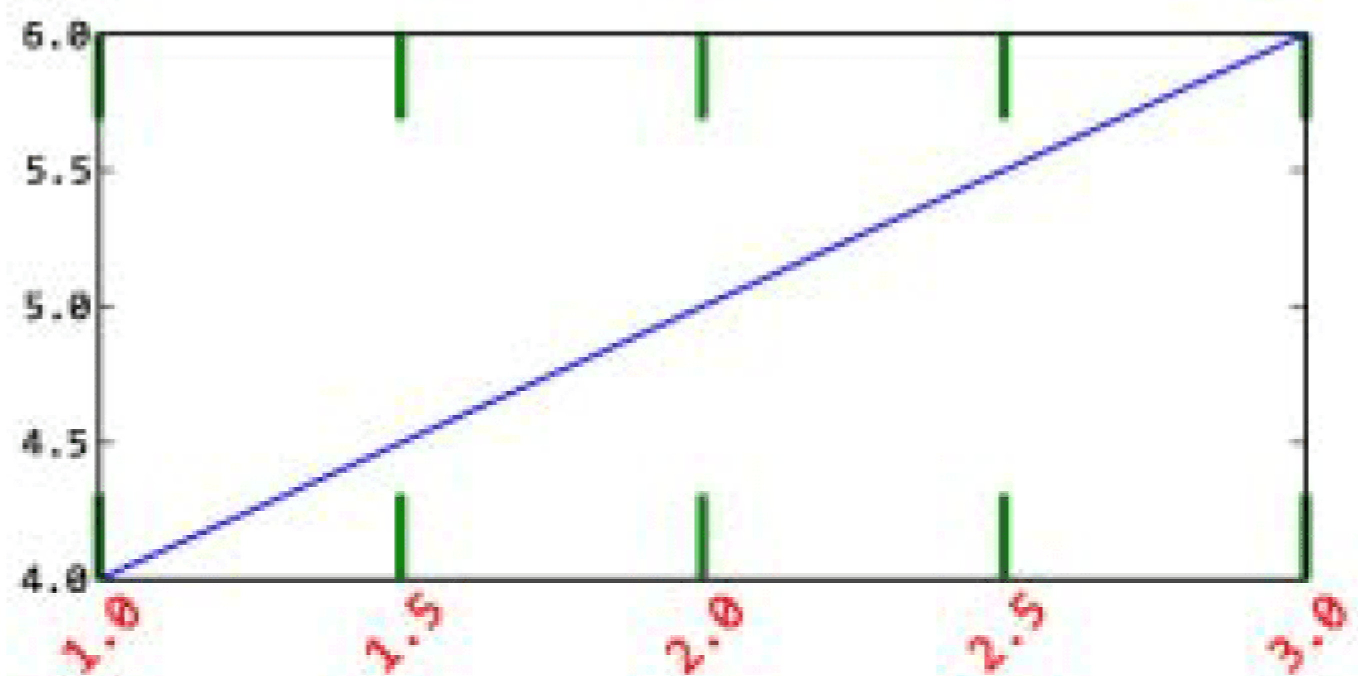


图 5-7　手工配置 X 轴的刻度线和刻度文本

请注意，这个例子只是为了演示 Artist 对象的各种属性，实际上使用 pyplot 模块中的 xticks() 能够更快地完成 X 轴上刻度标签的配置：

```
>>> plt.xticks(fontsize=16, color="red", rotation=45)
```

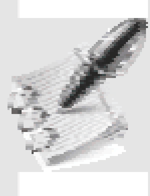
xticks()只能设置刻度标签的属性，不能设置刻度线的属性。感兴趣的读者可以在 IPython 中输入“plt.xticks??”查看其源代码。

上面的例子中，副刻度线列表为空，这是因为用于计算副刻度位置的对象默认为 NullLocator，它不产生任何刻度线。而计算主刻度位置的对象为 AutoLocator，它会根据当前的缩放等配置自动计算刻度的位置：

```
>>> axis.get_minor_locator() # 计算副刻度位置的对象
```

```
<matplotlib.ticker.NullLocator instance at 0x0A014300>
>>> axis.get_major_locator() # 计算主刻度位置的对象
<matplotlib.ticker.AutoLocator instance at 0x09281B20>
```

matplotlib 提供了多种配置刻度线位置的 Locator 类，以及控制刻度标签显示的 Formatter 类。下面的程序设置 X 轴的主刻度为 $\pi/4$，副刻度为 $\pi/20$，并且主刻度上的标签用数学符号显示 π。程序的输出如图 5-8 所示。



matplotlib_axis_text.py
自定义坐标轴的刻度和文字

```
import matplotlib.pyplot as pl
from matplotlib.ticker import MultipleLocator, FuncFormatter ❶
import numpy as np
x = np.arange(0, 4*np.pi, 0.01)
y = np.sin(x)
pl.figure(figsize=(8,4))
pl.plot(x, y)
ax = pl.gca()

def pi_formatter(x, pos): ❷
    """
    比较啰嗦地将数值转换为以 pi/4 为单位的刻度文本
    """
    m = np.round(x / (np.pi/4))
    n = 4
    while m!=0 and m%2==0: m, n = m//2, n//2
    if m == 0:
        return "0"
    if m == 1 and n == 1:
        return "$\pi$"
    if n == 1:
        return r"$%d \pi$" % m
    if m == 1:
        return r"$\frac{\pi}{%d}$" % n
    return r"$\frac{%d \pi}{%d}$" % (m,n)

# 设置两个坐标轴的范围
pl.ylim(-1.5,1.5)
pl.xlim(0, np.max(x))

# 设置图的底边距
pl.subplots_adjust(bottom = 0.15)
```

```
pl.grid() #开启网格

# 主刻度为pi/4
ax.xaxis.set_major_locator( MultipleLocator(np.pi/4) ) ❸

# 主刻度文本用 pi_formatter 函数计算
ax.xaxis.set_major_formatter( FuncFormatter( pi_formatter ) ) ❹

# 副刻度为pi/20
ax.xaxis.set_minor_locator( MultipleLocator(np.pi/20) ) ❺

# 设置刻度文本的大小
for tick in ax.xaxis.get_major_ticks():
    tick.label1.set_fontsize(16)
pl.show()
```

❶与刻度定位和文本格式化相关的类都在 matplotlib.ticker 模块中定义，程序从中载入了如下两个类：

- MultipleLocator：❸❺以指定值的整数倍为刻度放置主、副刻度线。
- FuncFormatter：❹使用指定的函数计算刻度文本，它会将刻度值和刻度的序号作为参数传递给计算刻度文本的函数。❷程序中通过 pi_formatter()计算出刻度值对应的刻度文本。

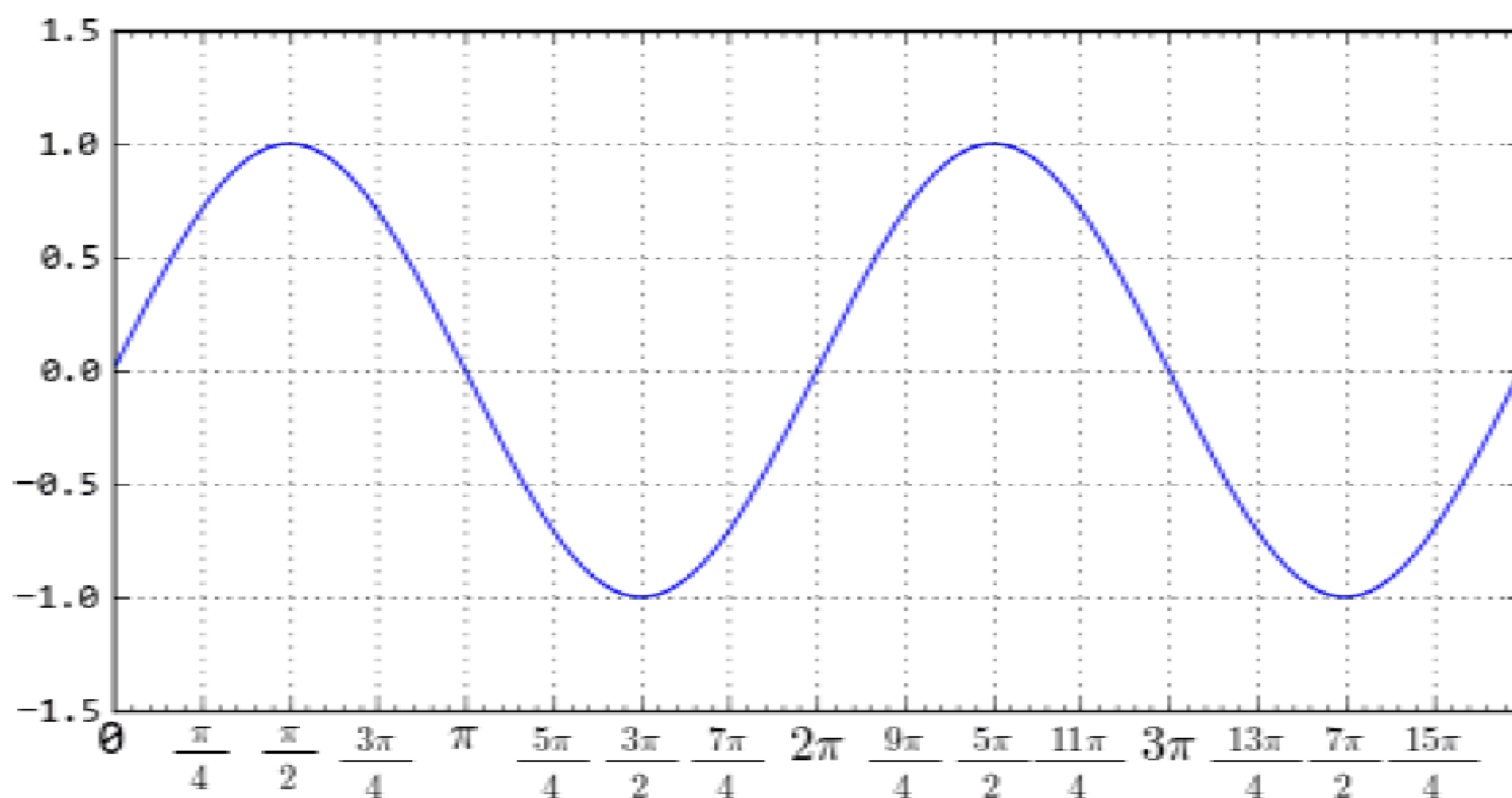


图 5-8　配置 X 轴上刻度线的位置和文本，并开启副刻度线

5.2.5 Artist 对象的关系

为了方便读者理解图表中各种 Artist 对象之间的关系，本书提供了一个输出 Artist 对象关系图的小程序。



graphviz_matplotlib.py

将 Artist 对象的关系以 Graphviz 格式的图形语言输出

“graphviz_matplotlib.py”中的 graphviz()能将 Artist 对象的内部关系输出成 Graphviz 格式的图形语言：

```
>>> from graphviz_matplotlib import graphviz
>>> print graphviz(plt.gcf())
digraph structs {
...
```

为了将图形语言转换成实际的关系图，读者可以从 Graphviz 的官方网站下载 Graphviz 软件包，或者使用 Graphviz 的在线编辑器。



http://www.graphviz.org 和 http://graph.gafol.net

Graphviz 的官方网站和在线编辑器

下面我们看一个例子。对于由如下程序产生的 Figure 对象，调用“graphviz(plt.gcf())”将得到如图 5-9 所示的关系图[①]。

```
plt.figure()
plt.subplot(211)
plt.bar([1,2,3],[1,2,3])
plt.subplot(212)
plt.plot([1,2,3])
```

图 5-9 中以灰色填充的矩形表示列表，其他的矩形表示各种 Artist 对象。Artist 对象之间的关系使用带箭头的细线表示，细线旁边的文本为属性名。

例如，从 Figure 矩形到 Rectangle 矩形的箭头表示 Figure 对象的 patch 属性是一个 Rectangle 对象，而 Figure 对象的 axes 属性是一个含有两个元素的列表，每个元素都是一个 AxesSubplot 对象。请读者仔细观察图 5-9，并在 IPython 中输入相应的语句以确认各个 Artist 对象之间的关系。

① “graphviz_matplotlib.py”中有几个设置关系图输出的选项，它们决定了哪些 Artist 对象的哪些属性将被展开显示。



图 5-9 图表中各个 Artist 对象的关系

5.3 坐标变换和注释

我们可以为图表添加文字、箭头以及各种标注元素，对图表中的重点区域进行说明。下面先看一个例子：



matplotlib_annotation.py
为图表添加各种注释元素

```
import numpy as np
import matplotlib.pyplot as plt

def func1(x): ❶
    return 0.6*x + 0.3

def func2(x): ❶
    return 0.4*x*x + 0.1*x + 0.2

def find_curve_intersects(x, y1, y2):
    d = y1 - y2
    idx = np.where(d[:-1]*d[1:]<=0)[0]
    x1, x2 = x[idx], x[idx+1]
    d1, d2 = d[idx], d[idx+1]
    return -d1*(x2-x1)/(d2-d1) + x1

x = np.linspace(-3,3,100) ❷
f1 = func1(x)
f2 = func2(x)
plt.figure(figsize=(8,4))
plt.plot(x, f1)
plt.plot(x, f2)

x1, x2 = find_curve_intersects(x, f1, f2) ❸
plt.plot(x1, func1(x1), "o")
plt.plot(x2, func1(x2), "o")

plt.fill_between(x, f1, f2, where=f1>f2, facecolor="green", alpha=0.5) ❹

from matplotlib import transforms
ax = plt.gca()
trans = transforms.blended_transform_factory(ax.transData, ax.transAxes)
plt.fill_between([x1, x2], 0, 1, transform=trans, alpha=0.1) ❺

a = plt.text(0.05, 0.95, u"直线和二次曲线的交点",  ❻
    transform=ax.transAxes,
    verticalalignment = "top",
    fontsize = 18,
    bbox={"facecolor":"red","alpha":0.4,"pad":10}
)

arrow = {"arrowstyle":"fancy,tail_width=0.6",
        "facecolor":"gray",
        "connectionstyle":"arc3,rad=-0.3"}

plt.annotate(u"交点", ❼
    xy=(x1, func1(x1)), xycoords="data",
    xytext=(0.05, 0.5), textcoords="axes fraction",
```

```
        arrowprops = arrow)

    plt.annotate(u"交点", ❼
        xy=(x2, func1(x2)), xycoords="data",
        xytext=(0.05, 0.5), textcoords="axes fraction",
        arrowprops = arrow)

    xm = (x1+x2)/2
    ym = (func1(xm) - func2(xm))/2+func2(xm)
    o = plt.annotate(u"直线大于曲线区域", ❼
        xy =(xm, ym), xycoords="data",
        xytext = (30, -30), textcoords="offset points",
        bbox={"boxstyle":"round", "facecolor":(1.0, 0.7, 0.7), "edgecolor":"none"},
        fontsize=16,
        arrowprops={"arrowstyle":"->"}
    )
    plt.show()
```


程序的输出如图 5-10 所示，图中演示了下面所列出的标注效果：

- 用两个小圆点表示直线和曲线的两个交点。
- 对在两个交点之间且位于直线和曲线之间的面积进行了填充。
- 使用一个高为整个子图高度、左右边位于两个交点的矩形来表示两个交点之间的区间。
- 在图的左上角放置了说明文字。
- 对两个交点和填充面积使用了带箭头的注释说明。

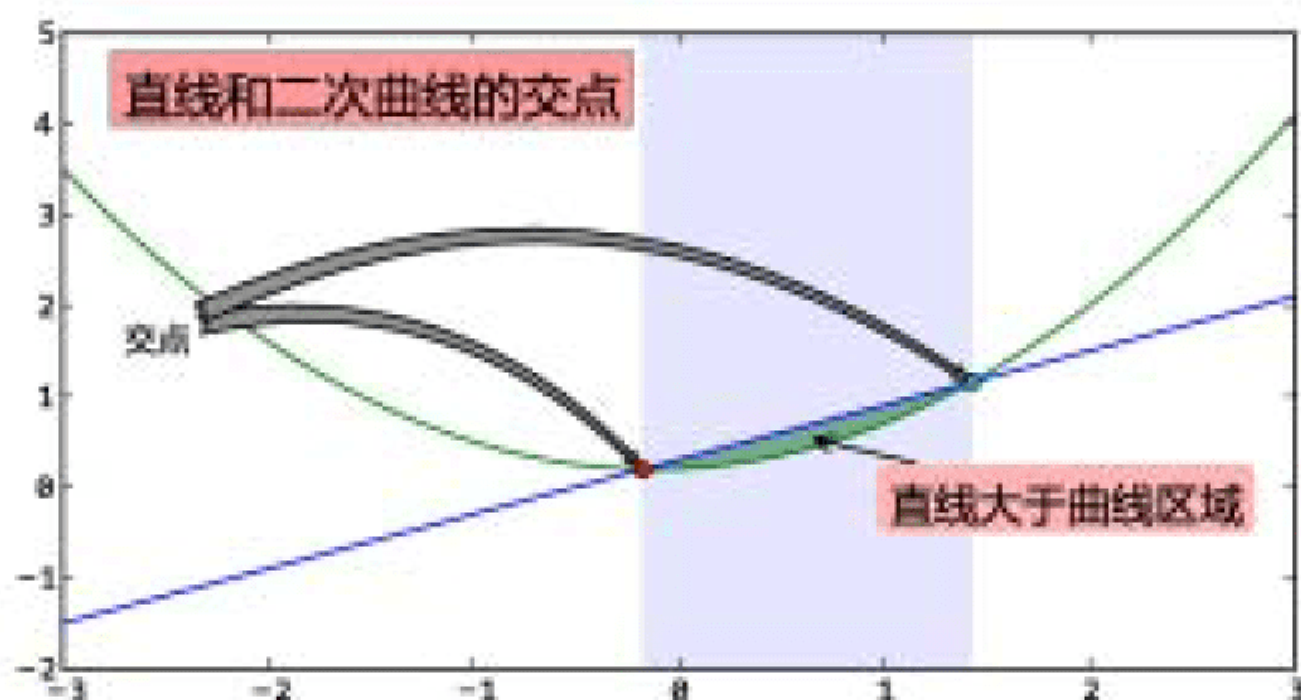


图 5-10　为图表添加各种注释元素

首先，❶定义了两个函数 func1 和 func2，它们分别是计算一条直线和一条二次曲线的函数。然后❷计算这两个函数在区间(-3,3)上的值，并且调用 plot()绘制成曲线图。

❸为了标出两个交点，我们用 find_curve_intersects()计算两条曲线 f1 和 f2 的交点所对应的 X 轴坐标 x1 和 x2。交点处的小圆点仍然使用 plot()进行绘制，这时所传递的 X-Y 轴的数据为单一数值，并且以'o'为样式进行绘图。

如何计算两条曲线的交点

当两条曲线的 Y 轴坐标值 y1 和 y2 使用相同的 X 轴坐标数组 x 计算时，很容易计算它们的交点。首先计算两条曲线在 Y 轴的差值 d=y1-y2，然后找到符号相反的两个连续的差值的下标 idx 和 idx+1。计算直线(x[idx],d[idx])-(x[idx+1],d[idx+1])和 X 轴的交点就可得到两条曲线交点的 X 轴坐标 xc。如果要计算交点的 Y 轴坐标，只需要调用 np.interp(xc, x, y1)对曲线进行线性插值即可。

❹接下来调用 fill_between()绘制 X 轴上在两个交点之间、Y 轴上在两条曲线之间的面积部分，并通过 facecolor 和 alpha 参数指定填充的颜色和透明度。fill_between()的调用参数如下：

```
fill_between(x, y1, y2=0, where=None)
```

其中，x 参数是长度为 N 的数组，y1 和 y2 参数是长度为 N 的数组或单个数值。当 y1 或 y2 为单个数值时，它们相当于一个长度为 N、元素数值都相同的数组。fill_between()将填充 Y 轴在 y1 和 y2 之间的部分。如果 where 参数为 None，将对数组 x 中的所有元素进行填充；如果 where 是一个布尔数组，就只填充其中 True 所对应的数组 x 的元素。程序中数组 x 的取值范围为(-3,3)，由于设置了条件“where=f1>f2”，因此仅绘制直线在二次曲线之上的部分。

❺绘制 X 轴上在两个交点之间的矩形区域；❻用 text()在图表中添加说明文字；❼最后用 annotate()为图表添加了三个带箭头的注释。

为了真正理解程序的细节，首先需要了解 matplotlib 中坐标变换的工作原理。

5.3.1　4 种坐标系

在使用 matplotlib 绘制的一幅图表中，有 4 种坐标系：

- 数据坐标系：它是描述数据空间中位置的坐标系，例如对于图 5-10，它的数据坐标系为 X 轴在(-3,3)之间、Y 轴在(-2,5)之间。
- 子图坐标系：描述子图中位置的坐标系，子图的左下角坐标为(0,0)，右上角坐标为(1,1)。
- 图表坐标系：一幅图表可以包含多个子图，并且子图周围都有一定的余白，因此还需要用图表坐标系描述图表显示区域的某个点，图表的左下角坐标为(0,0)，右上角坐标为(1,1)。
- 窗口坐标系：它是绘图窗口中以像素为单位的坐标系。左下角坐标为(0, 0)，右上角坐标为(width,height)，其中的 width 和 height 分别是以像素为单位的绘图窗口的内宽和内高，即不包括标题栏、工具条以及状态栏等部分。

Axes 对象的 transData 属性是数据坐标变换对象，transAxes 属性是子图坐标变换对象。Figure 对象的 transFigure 属性是图表坐标变换对象。

通过上述坐标变换对象的 transform()方法，可以将此坐标系下的坐标转换为窗口坐标系中的坐标。下面看一个例子，首先在 IPython 中运行“matplotlib_annotation.py”演示程序：

```
>>> run matplotlib_annotation.py
```

下面的程序计算数据坐标系中的坐标点(-3,-2)和(3,5)在绘图窗口中的坐标：

```
>>> type(ax.transData)
<class 'matplotlib.transforms.CompositeGenericTransform'>
>>> ax.transData.transform([(-3,-2),(3,5)])
array([[  80.,   32.],
       [ 576.,  288.]])
```

下面的程序计算子图坐标系中的坐标点(0,0)和(1,1)在绘图窗口中的位置，得到的结果和上面的程序相同。也就是说，子图的左下角坐标(0,0)和数据坐标系中的坐标(-3,-2)在屏幕上是同一个点。观察图 5-10 就可以知道这显然是正确的。

```
>>> ax.transAxes.transform([(0,0),(1,1)])
array([[  80.,   32.],
       [ 576.,  288.]])
```

最后计算图表坐标系中坐标点(0,0)和(1,1)在绘图窗口中的位置，可以看出绘图区域的宽度为 640 个像素、高度为 320 个像素：

```
>>> plt.gcf().transFigure.transform([(0,0),(1,1)])
array([[   0.,    0.],
       [ 640.,  320.]])
```

通过坐标变换对象的 inverted()方法，可以获得它的逆变换对象。例如下面的程序计算绘图窗口中的坐标点(320,160)在数据坐标系中的坐标，结果为(-0.09677419,1.5)：

```
>>> inv = ax.transData.inverted()
>>> type(inv)
<class 'matplotlib.transforms.CompositeGenericTransform'>
>>> inv.transform((320, 160))
array([-0.09677419,  1.5       ])
```

请读者仔细观察程序输出的图表，子图的上下余白相同，而左侧余白略大于右侧余白，因此绘图区域的中心点(320,160)并不是数据区域的中心点(0, 1.5)。

当调用 xlim()修改子图显示的 X 轴范围之后，它的数据坐标变换对象也同时发生变化：

```
>>> plt.xlim(-3,2) # 设置X轴的范围为-3到2
(-3, 2)
>>> ax.transData.transform((3,5)) # 数据坐标变换对象已经发生变化
array([ 675.2,  288. ])
>>> plt.draw() # 刷新绘图区域，显示新的绘图范围
```

下面我们回头看看图 5-10 中绘制矩形区间的程序：

```
from matplotlib import transforms
ax = plt.gca()
trans = transforms.blended_transform_factory(ax.transData, ax.transAxes)
plt.fill_between([x1, x2], 0, 1, transform=trans, alpha=0.1)
```

矩形区间使用 fill_between()进行绘制。由于所绘制矩形的左右两边要始终经过两个交点，因此矩形的 X 轴坐标必须使用数据坐标系中的坐标：x1 和 x2。又由于矩形的高度始终充满整个子图的高度，因此矩形的 Y 轴坐标必须是子图坐标系中的坐标：0 和 1。

程序中，使用 blended_transform_factory()创建这种混合坐标系。它的两个参数都是坐标变换对象，它从第一个参数获得 X 轴的坐标变换，从第二个参数获得 Y 轴的坐标变换。因此它所返回的坐标变换对象 trans 的 X 轴使用数据坐标系，而 Y 轴使用子图坐标系。程序中，将混合坐标变换对象 trans 传递给 fill_between()的 transform 参数，这样绘制的填充区域就能始终保持左右两边通过两个交点，而上下两边位于子图边框之上。

5.3.2 坐标变换的步骤

从一个坐标系变换到另一个坐标系，中间需要经过几个步骤。而且数据坐标系不一定是笛卡尔坐标系，它可能是极坐标系或对数坐标系。因此坐标系的变换并不是一个简单的二维仿射变换(2D Affine Transformation)。下面从最简单的图表坐标变换对象 transFigure 开始，介绍 matplotlib 的坐标变换是如何进行的。

```
>>> fig = plt.gcf()
>>> fig.transFigure
BboxTransformTo(
        TransformedBbox(
            Bbox(array([[ 0.,  0.],
                       [ 8.,  4.]])),
            Affine2D(array([[ 80.,   0.,   0.],
                           [  0.,  80.,   0.],
                           [  0.,   0.,   1.]]))
        )
    )
```

这个坐标变换对象的内容有些复杂，它是一个 BboxTransformTo 对象，其中包含一个 TransformedBbox 对象，而 TransformedBbox 对象又包含一个 Bbox 对象和一个 Affine2D 对象。

- Bbox：定义一个矩形区域——[[x0, y0], [x1, y1]]。在本例中，矩形的两个顶点坐标分别为(0,0)和(8,4)，它是窗口的英寸大小，通过 figsize 参数传递给 figure()。
- Affine2D：二维仿射变换对象，它是一个矩阵，通过它和齐次向量进行乘积得到变换之后的坐标，由于矩阵中只有对角线上的值不为零，因此仿射变换只进行缩放变换。它将坐标(x,y)变换为(80*x, 80*y)。

仿射变换

二维空间的仿射变换矩阵的大小为 3*3，为了进行仿射变换需要使用齐次坐标，即用三维向量(x,y,1)表示二维平面上的点(x,y)。仿射变换就是仿射矩阵和向量的乘积。由于变换矩阵最下一行的数值始终是(0, 0, 1)，因此有时也将它写成 2*3 的矩阵形式。

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- TransformedBbox：对矩形区域进行仿射变换之后得到一个新的矩形区域。在本例中，所得到矩形区域的两个顶点为(0,0)和(640,320)。它正好是以像素点为单位的窗口的大小，因此仿射变换矩阵中的数值 80 实际上是 Figure 对象的 dpi 属性：

```
>>> fig.dpi
80
```

- BboxTransformTo：它是一个从单位矩形区域转换到指定矩形区域的变换。在本例中，它是一个将矩形区域(0,0)-(1,1)变换到矩形区域(0,0)-(640,320)的坐标变换对象，因此它能将坐标从图表坐标系转换为窗口坐标系中的坐标。

下面的程序查看组成变换的各个部分：

```
>>> fig.transFigure._boxout
TransformedBbox(...)
>>> fig.transFigure._boxout.bounds          #仿射变换后的矩形区域
(0.0, 0.0, 640.0, 320.0)
>>> fig.transFigure._boxout._bbox          #以英寸为单位的矩形区域
Bbox(array([[ 0.,  0.], [ 8.,  4.]]))
>>> fig.transFigure._boxout._transform          #仿射变换对象
Affine2D(array([[ 80.,   0.,   0.],
               [  0.,  80.,   0.],
               [  0.,   0.,   1.]]))
```

实际上，fig.transFigure 中的仿射变换对象可以通过 fig.dpi_scale_trans 获得：

```
>>> fig.dpi_scale_trans == fig.transFigure._boxout._transform
True
```

接下来查看子图坐标变换对象的内容：

```
>>> ax.transAxes
BboxTransformTo(
    TransformedBbox(
        Bbox(array([[ 0.125,  0.1  ],[ 0.9  ,  0.9  ]])),
```

```
            ...省略... # 这一部分实际上是 fig.transFigure 的内容
        )
    )
```

我们看到 ax.transAxes 是一个 BboxTransformTo 对象，因此它可以将(0,0)-(1,1)区域变换为另外一个区域。而此区域是一个 TransformedBbox 对象，它是将矩形区域(0.125,0.1)-(0.9,0.9)通过 fig.transFigure 变换之后得到的区域。因此在 ax.transAxes 对象内部使用了 fig.transFigure 变换：

```
>>> ax.transAxes._boxout._transform == fig.transFigure
True
```

而此变换中的矩形区域(0.125, 0.1)-(0.9, 0.9)，实际上是子图在图表坐标系中的位置：

```
>>> ax.get_position()
Bbox(array([[ 0.125,  0.1  ], [ 0.9  ,  0.9  ]]))
```

子图在窗口坐标系中的矩形区域为：

```
>>> ax.transAxes._boxout.bounds
(80.0, 31.999999999999993, 496.0, 256.0)
```

因此，ax.transAxes 实际上是一个将矩形区域(0,0)-(1,1)变换到矩形区域(80.0,32)-(496.0, 256.0)的坐标变换对象。

最后我们观察一下数据坐标系的变换对象 ax.transData。由于 ax.transData 由 ax.transScale、ax.transLimits 和 ax.transAxes 共同构成，因此先看看 ax.transLimits 和 ax.transScale 的内容。transLimits 是一个 BboxTransformFrom 对象，它是一个将指定矩形区域变换为(0,0)-(1,1)矩形区域的变换对象。

```
>>> ax.transLimits
BboxTransformFrom(
    TransformedBbox(
        Bbox(array([[-3., -2.], [ 3.,  5.]])),
        TransformWrapper(BlendedAffine2D(IdentityTransform(),IdentityTransform()))
        )
)
```

而 transLimits 的源矩形区域为一个 TransformedBbox 对象，它是一个将矩形区域(-3,-2)-(3,5)通过坐标变换之后得到的矩形区域。而此处的变换由如下程序定义，它实际上是一个恒等变换：

```
TransformWrapper(BlendedAffine2D(IdentityTransform(),IdentityTransform()))
```

因此 transLimits 的最终效果就是将矩形区域(-3,-2)-(3,5)变换为矩形区域(0,0)-(1,1)：

```
>>> ax.transLimits.transform((-3,-2))
array([ 0.,  0.])
>>> ax.transLimits.transform((3,5))
array([ 1.,  1.])
```

而矩形区域(-3,-2)-(3,5)实际上由 X 轴和 Y 轴的显示范围决定：

```
>>> ax.get_xlim() # 获得 X 轴的显示范围
(-3.0, 3.0)
>>> ax.get_ylim() # 获得 Y 轴的显示范围
(-2.0, 5.0)
```

由于 transLimits 将数据坐标系的显示范围变换为单位矩形，而 transAxes 将单位矩形变换为以像素为单位的窗口矩形范围，因此这两个变换的综合效果就是将数据坐标变换为窗口坐标。我们可以用“+”号将两个变换连接起来创建一个新的变换对象，例如“ax.transLimits + ax.transAxes”表示先进行 ax.transLimits 变换，然后进行 ax.transAxes 变换，变换对象就像流水线上生产产品一样，一步一步地对坐标点进行变换。下面的程序比较它和 ax.transData 的变换结果：

```
>>> t = ax.transLimits + ax.transAxes
>>> t.transform((0,0))
array([ 328.        ,  105.14285714])
>>> ax.transData.transform((0,0))
array([ 328.        ,  105.14285714])
```

实际上，为了支持不同比例的坐标轴，transData 中还包括一个 transScale 变换，即“transData = transScale + transLimits + transAxes”。由于本例中，transScale 是一个恒等变换，因此“ax.transLimits + ax.transAxes”和 ax.transData 的变换效果一样：

```
>>> ax.transScale
TransformWrapper(BlendedAffine2D(IdentityTransform(),IdentityTransform()))
```

当使用 semilogx()、semilogy()以及 loglog()等绘图函数绘制对数坐标轴的图表时，或者使用 Axes 的 set_xscale()和 set_yscale()等方法将坐标轴设置为对数坐标时，transScale 就不再是恒等变换了：

```
>>> ax.set_xscale("log") # 将 X 轴改为对数坐标
>>> ax.transScale
TransformWrapper(BlendedGenericTransform(
    <matplotlib.scale.Log10Transform object at 0x013A7B50>,
    IdentityTransform()))
>>> ax.set_xscale("linear") # 将 X 轴改为线性坐标
```



由于本例中 X 轴的取值范围是(-3,3)，因此如果将 X 轴改为对数坐标，并且重新绘图的话，会产生很多错误信息。

5.3.3 制作阴影效果

下面用我们学到的坐标变换的知识绘制一个带阴影效果的曲线。完整的程序如下，效果如图 5-11 所示。



matplotlib_shadow.py
通过坐标变换为曲线添加阴影

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.transforms as transforms

x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)

N = 7 # 阴影的条数
for i in xrange(N, 0, -1):
    offset = transforms.ScaledTranslation(i, -i, transforms.IdentityTransform()) ❶
    shadow_trans = plt.gca().transData + offset ❷
    plt.plot(x,y,linewidth=4,color="black",
        transform=shadow_trans,  ❸
        alpha=(N-i)/2.0/N)

plt.plot(x,y,linewidth=4,color='black')
plt.ylim((-1.5, 1.5))
plt.show()
```

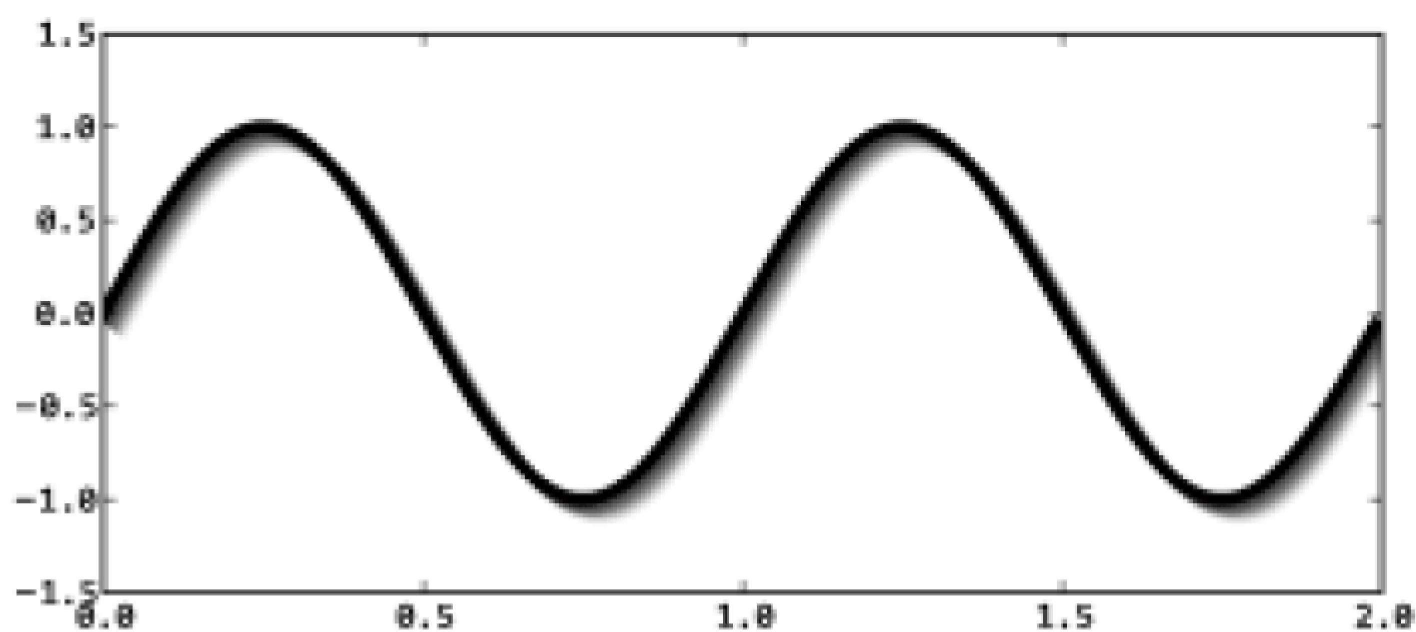


图 5-11　使用坐标变换绘制的带阴影的曲线

程序中，首先使用循环绘制 N 条透明度和偏移量逐渐变化的曲线，然后再绘制实际的曲线以实现阴影效果。

❶offset 是一个 ScaledTranslation 对象，它的第一个和第二个参数决定了 X 轴和 Y 轴的偏移量，而第三个参数是一个坐标变换对象，经过它变换之后，再进行偏移变换。由于程序中的第三个参数是一个恒等变换，因此 offset 实际上是一个单纯的偏移变换：对 X 轴坐标增加 i，对 Y 轴坐标减少 i。

可以通过 IPython 查看最后的 i 为 1 时的 offset：

```
>>> run matplotlib_shadow.py
>>> offset.transform((0,0)) # 将(0,0)变换为(1,-1)
array([ 1., -1.])
```

❷阴影曲线的坐标变换由 shadow_trans 完成，它由数据坐标变换对象 transData 和 offset 组成：

```
>>> plt.gca().transData.transform((0,0)) # 对(0,0)进行数据坐标变换
array([  80.,  240.])
>>> shadow_trans.transform((0,0)) # 对(0,0)进行数据坐标变换和偏移变换
array([  81.,  239.])
```

❸最后通过 transform 参数将 shadow_trans 传递给 plot()进行绘图。由于 shadow_trans 是在完成数据坐标到窗口坐标的变换之后、再进行偏移变换的，因此无论当前的缩放比例如何，阴影效果将始终保持一致。

5.3.4 添加注释

pyplot 模块中提供了两个绘制文字的函数：text()和 figtext()。它们分别调用当前 Axes 对象和当前 Figure 对象的 text()方法进行绘图。text()默认在数据坐标系中添加文字，而 figtext()则默认在图表坐标系中添加文字。可以通过 transform 参数改变文字所在的坐标系，下面的程序演示了如何在数据坐标系、子图坐标系以及图表坐标系中添加文字。



matplotlib_text.py
在数据坐标系、子图坐标系以及图表坐标系中添加文字

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-1,1,10)
y = x**2

fig = plt.figure(figsize=(8,4))
ax = plt.subplot(111)
```

```
plt.plot(x,y)

for i, (_x, _y) in enumerate(zip(x, y)):
    plt.text(_x, _y, str(i), color="red", fontsize=i+10) ❶

plt.text(0.5, 0.8, u"子图坐标系中的文字", color="blue", ha="center",
    transform=ax.transAxes) ❷

plt.figtext(0.1, 0.92, u"图表坐标系中的文字", color="green") ❸

plt.show()
```

❶由于没有设置 transform 参数，text()默认将在数据坐标系中创建文字，这里通过 fontsize 参数修改文字的大小。❷通过 transform 参数将文字的坐标变换改为 ax.transAxes，因此文字在子图坐标系中。ha 参数为'center'表示坐标点(0.5, 0.8)在水平方向上是文字的中心，ha 是 horizontal alignment 的缩写，含义是水平对齐。❸调用 figtext()在图表坐标系中添加文字。

程序的输出如图 5-12 所示。请读者使用缩放和平移工具改变子图的显示范围，会发现数据坐标系中的文字将跟随曲线一起移动，而其他两个坐标系中的文字位置不变。单击绘图窗口工具栏中的倒数第二个图标按钮，打开“Subplot Configuration Tool”对话框，调节 top、right、bottom 和 left 等参数，会发现子图坐标系中的文字也会跟着改变位置，但水平方向上它和子图的中心始终保持一致。而图表坐标系中文字的位置，只有在改变窗口大小时才会发生变化。

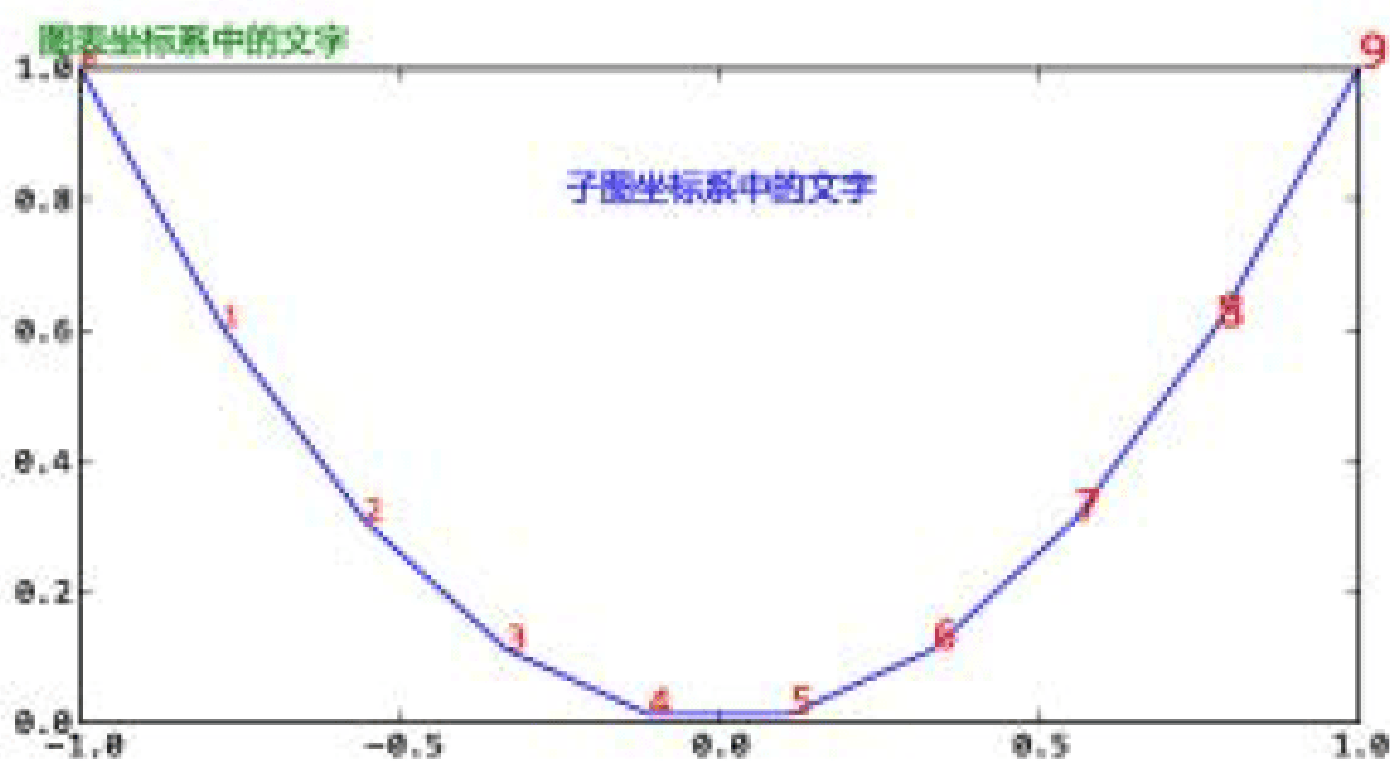


图 5-12　三个坐标系中的文字

绘制文字的函数还有许多关键字参数，可以设置文字、外框的样式，请读者参考 matplotlib 的用户手册，这里就不再详细介绍了。

通过 pyplot 模块的 annotate()可以绘制带箭头的注释文字。annotate()的调用格式如下：

```
annotate(s, xy, xytext=None, xycoords='data', textcoords='data', arrowprops=None, ...)
```

其中，s 参数是注释文本，xy 是箭头所指位置的坐标，xytext 是注释文本所在的坐标，xycoords

和 textcoords 分别指定箭头坐标和注释文本坐标的坐标变换方式。

带箭头的注释需要指定两个坐标：箭头所指位置的坐标和注释文字所在的坐标。而这两个坐标可以使用不同的坐标变换方法转换为窗口坐标。xycoords 和 textcoords 参数的值都是字符串，它们可以有表 5-2 所示的几种选项：

表 5-2 xycoords 和 textcoords 参数的属性值

属 性 值	坐标变换方式
figure points	以点为单位，相对于图表左下角的坐标
figure pixels	以像素为单位，相对于图表左下角的坐标
figure fraction	图表坐标系中的坐标
axes points	以点为单位，相对于子图左下角的坐标
axes pixels	以像素为单位，相对于子图左下角的坐标
axes fraction	子图坐标系中的坐标
data	数据坐标系中的坐标
offset points	以点为单位，相对于点 xy 的坐标
polar	数据坐标系中的极坐标

其中，'figure fraction'、'axes fraction'和'data'分别表示使用图表坐标系、子图坐标系和数据坐标系中的坐标变换对象。由于图表和子图坐标系都是正规化之后的坐标系，使用起来不太方便，因此对于图表和子图还分别提供了以点和像素为单位的坐标变换方式。点和像素的单位类似，但是它不会随图表的 dpi 属性值而发生变化，它始终以每英寸 72 个点进行计算。

上述几种坐标变换都以固定的点为原点进行变换，有时我们希望以距离箭头的偏移量指定文字的坐标，这时可以使用'offset points'选项。

在图 5-10 中，所有注释的箭头坐标都采用'data'，因此无论如何放大或平移绘图区域，箭头都始终指向数据坐标系中的固定点。而注释文本“交点”的坐标变换方式采用'axes fraction'，因此“交点”始终保持在子图中的固定位置。而“直线大于曲线区域”注释文本的坐标采用'offset points'变换，因此文字和箭头的相对位置始终保持不变。

有兴趣的读者可以查看 matplotlib 中关于注释的坐标变换的源程序，从而更深入地研究坐标变换：

c:\Python26\Lib\site-packages\matplotlib\text.py
关于坐标变换的程序在此文件中的 Annotation._get_xy()中定义

最后，arrowprops 参数是一个描述箭头样式的字典。关于注释样式的详细配置请参考 matplotlib 的相关文档：

http://matplotlib.sourceforge.net/users/annotations_guide.html
matplotlib 官方网站中关于注释的文档

5.4 绘图函数简介

作为本章的最后一节，让我们看看如何用 matplotlib 绘制一些常用的图表。本节介绍的每个绘图函数都有许多关键字参数，用来设置图表的各种属性，由于篇幅有限，本书不能对其一一进行介绍。一般来说，如果读者需要对图表进行某种特殊的设置，都可以在绘图函数的说明文档或 matploblib 的演示页面中找到相关的说明。

5.4.1 对数坐标图

前面介绍过如何使用 plot()绘制曲线图，所绘制图表的 X-Y 轴坐标都是算术坐标。下面看看如何在对数坐标系中绘图。

绘制对数坐标图的函数有三个：semilogx()、semilogy()和 loglog()，它们分别绘制 X 轴为对数坐标、Y 轴为对数坐标以及两个轴都为对数坐标时的图表。



在对数坐标图中，因为轴上的刻度文本使用 LaTex 公式表示指数，所以图表的重绘比较费时。

下面的程序使用 4 种不同的坐标系绘制低通滤波器的频率响应曲线，结果如图 5-13 所示。其中，左上图为 plot()绘制的算术坐标系，右上图为 semilogx()绘制的 X 轴对数坐标系，左下图为 semilogy()绘制的 Y 轴对数坐标系，右下图为 loglog()绘制的双对数坐标系。使用双对数坐标系表示的频率响应曲线通常被称为波特图。



matplotlib_log.py
用 4 种不同的坐标系绘制低通滤波器的频率响应曲线

```
w = np.linspace(0.1, 1000, 1000)
p = np.abs(1/(1+0.1j*w)) # 计算低通滤波器的频率响应

plt.subplot(221)
plt.plot(w, p, linewidth=2)
plt.ylim(0,1.5)

plt.subplot(222)
plt.semilogx(w, p, linewidth=2)
plt.ylim(0,1.5)

plt.subplot(223)
plt.semilogy(w, p, linewidth=2)
plt.ylim(0,1.5)
```

```
plt.subplot(224)
plt.loglog(w, p, linewidth=2)
plt.ylim(0,1.5)
```

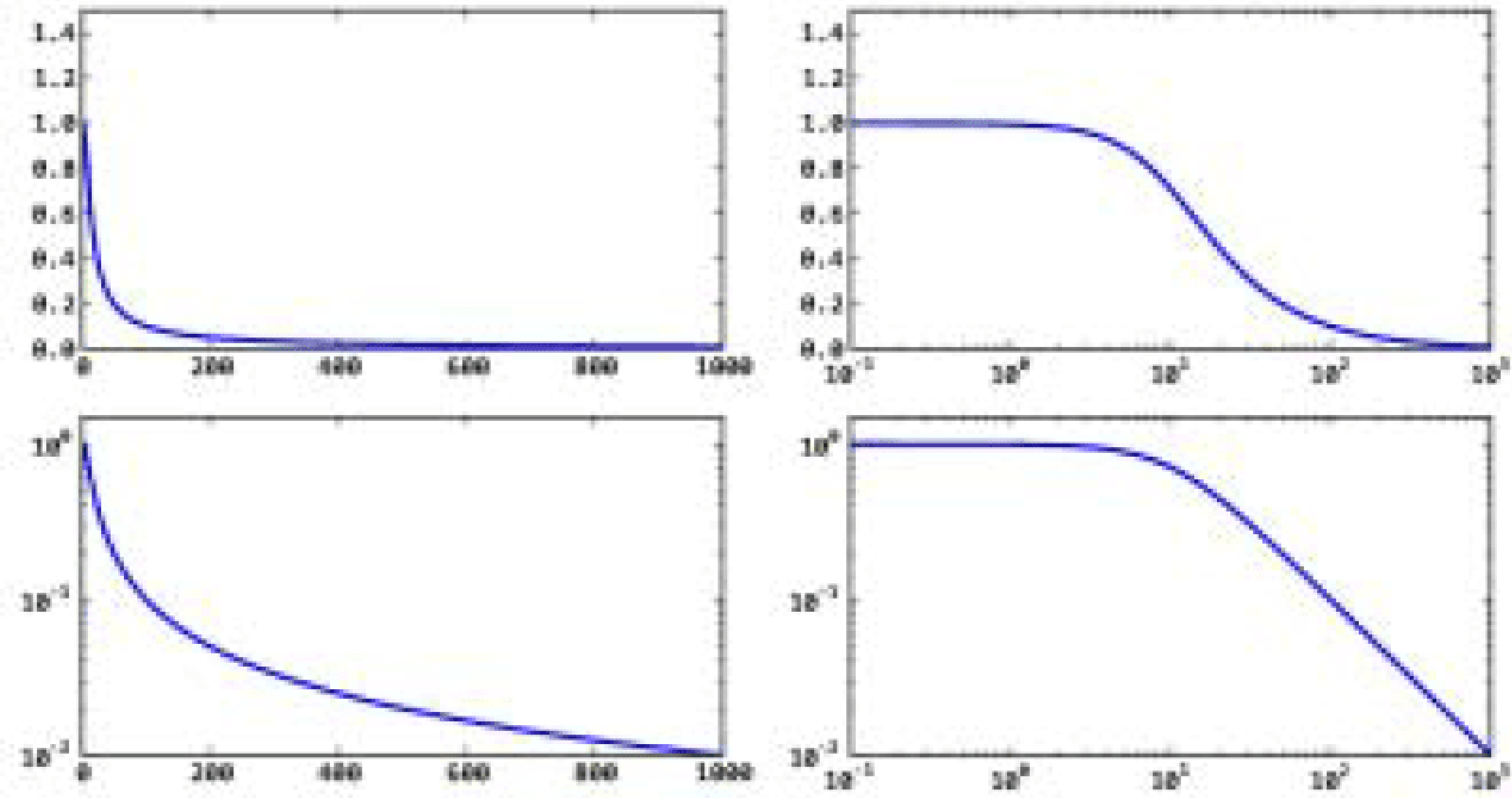
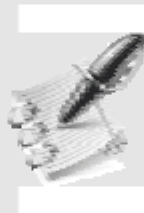


图 5-13　低通滤波器的频率响应曲线

5.4.2　极坐标图

极坐标系是和笛卡尔(X-Y)坐标系完全不同的坐标系，极坐标系中的点由一个夹角和一段相对中心点的距离来表示。下面的程序绘制极坐标图，效果如图 5-14 所示(见文前彩插)。


matplotlib_polar.py
在极坐标中绘图

```
theta = np.arange(0, 2*np.pi, 0.02)

plt.subplot(121, polar=True) ❶
plt.plot(theta, 1.6*np.ones_like(theta), linewidth=2) ❷
plt.plot(3*theta, theta/3, "--", linewidth=2)

plt.subplot(122, polar=True)
plt.plot(theta, 1.4*np.cos(5*theta), "--", linewidth=2)
plt.plot(theta, 1.8*np.cos(4*theta), linewidth=2)
plt.rgrids(np.arange(0.5, 2, 0.5), angle=45) ❸
plt.thetagrids([0, 45]) ❹
```

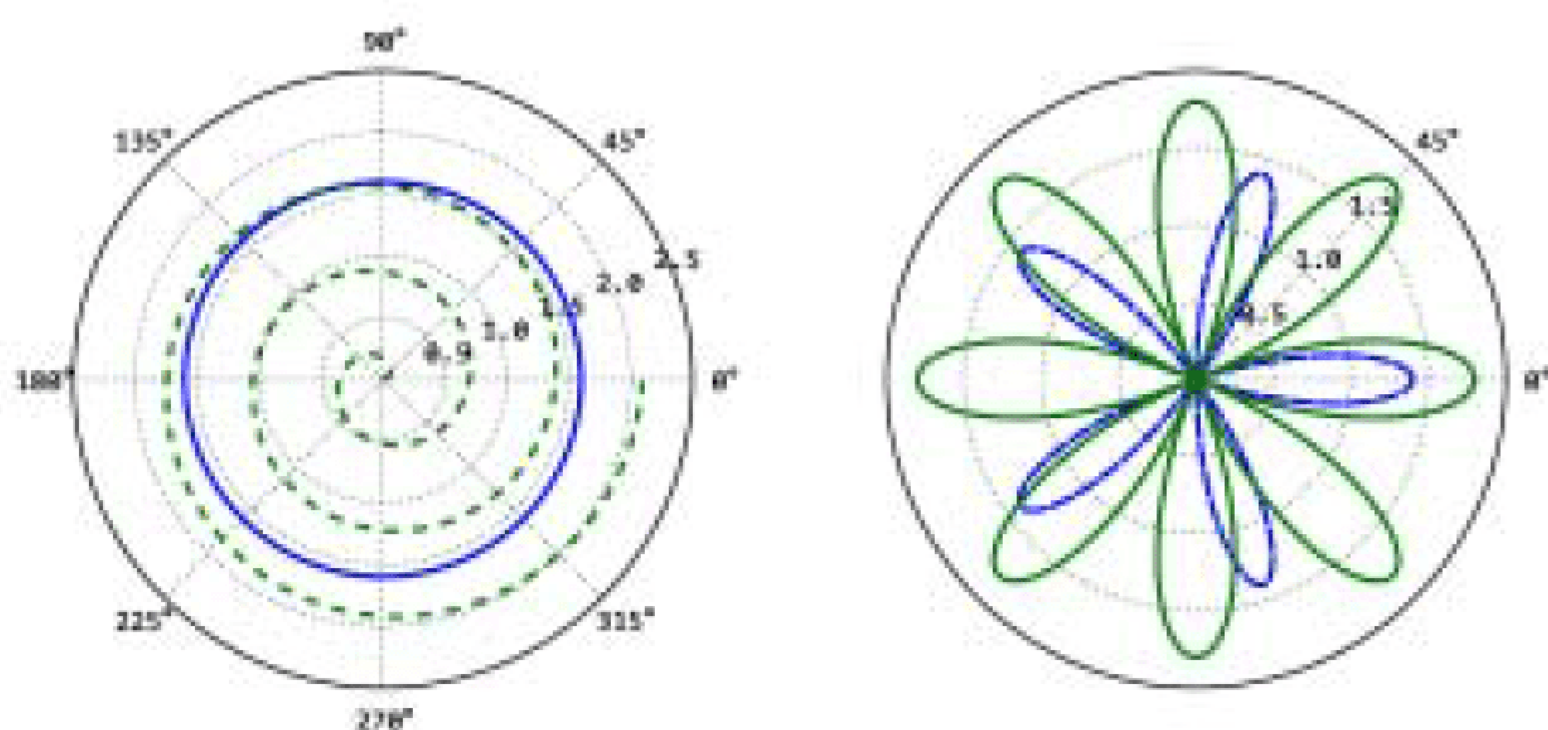


图 5-14　极坐标中的圆、螺旋线和玫瑰线

❶调用 subplot()创建子图时通过设置 polar 参数为 True，创建一个极坐标子图。❷然后调用 plot()在极坐标子图中绘图。也可以使用 polar()直接创建极坐标子图并在其中绘制曲线。

❸rgrids()设置同心圆栅格的半径大小和文字标注的角度。因此右图中的虚线圆圈有三个，半径分别为 0.5、1.0 和 1.5，这些文字沿着 45° 线排列。❹thetagrids()设置放射线栅格的角度，因此右图中只有两条放射线，角度分别为 0° 和 45° 。

5.4.3　柱状图

柱状图用其每根柱子的长度表示值的大小，它们通常用来比较两组或多组值。下面的程序从文件中读入中国人口的年龄分布数据[②]，并使用柱状图比较男性和女性的年龄分布，效果如图 5-15 所示(见文前彩插)。



matplotlib_bar.py
绘制比较男女人口的年龄分布图

```
data = np.loadtxt("china_population.txt")
width = (data[1,0] - data[0,0])*0.4 ❶
plt.figure(figsize=(8,5))
plt.bar(data[:,0]-width, data[:,1]/1e7, width, color="b", label=u"男") ❷
plt.bar(data[:,0], data[:,2]/1e7, width, color="r", label=u"女") ❸
plt.xlim(-width, 100)
plt.xlabel(u"年龄")
plt.ylabel(u"人口(千万)")
plt.legend()
```

② 人口分布数据由维基百科提供，仅供参考，不保证其正确性。

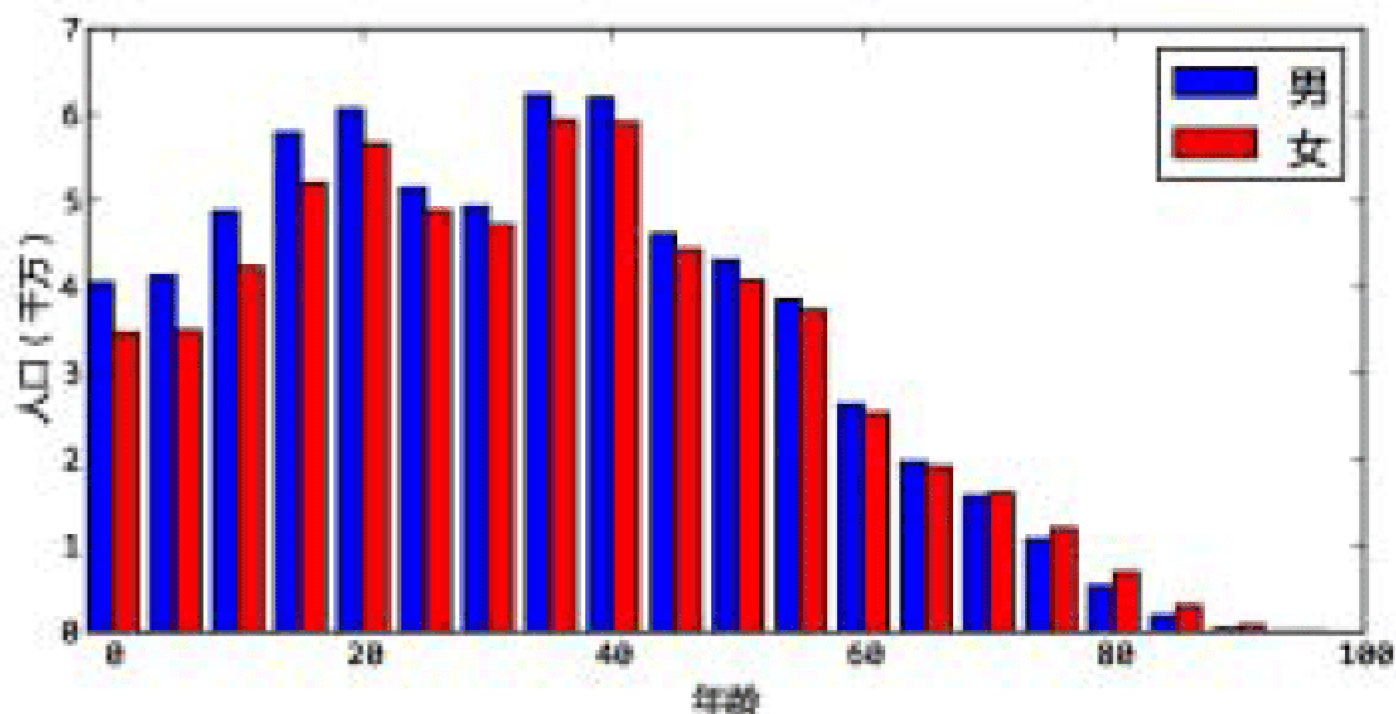


图 5-15　中国男女人口的年龄分布图

读入的数据中，第 0 列为年龄，它将作为柱状图的横坐标。❶首先计算柱状图中每根柱子的宽度，因为我们要在每个年龄段上绘制两根柱子，因此柱子的宽度应该小于年龄段的二分之一。这里以年龄段的 0.4 倍作为柱子的宽度。

❷调用 bar()绘制男性人口分布的柱状图。它的第一个参数为每根柱子左边缘的横坐标，为了让男性和女性的柱子以年龄刻度为中心，这里让每根柱子左侧的横坐标为“年龄减去柱子的宽度”。bar()的第二个参数为每根柱子的高度，第三个参数指定所有柱子的宽度。当第三个参数为序列时，可以为每根柱子指定宽度。

❸绘制女性人口分布的柱状图，这里以年龄为柱子的左边缘横坐标，因此女性和男性的人口分布图以年龄刻度为中心。由于 bar()不自动修改颜色，因此程序中通过 color 参数设置两个柱状图的颜色。

5.4.4　散列图

使用 plot()绘图时，如果指定样式参数为仅绘制数据点，那么所绘制的就是一幅散列图。例如：

```
>>> plt.plot(np.random.random(100), np.random.random(100), "o")
```

但是这种方法所绘制的点无法单独指定颜色和大小。而 scatter()所绘制的散列图却可以指定每个点的颜色和大小。下面的程序演示 scatter()的用法，效果如图 5-16 所示(见文前彩插)。


matplotlib_scatter.py
绘制散列图

```
plt.figure(figsize=(8,4))
x = np.random.random(100)
y = np.random.random(100)
plt.scatter(x, y, s=x*1000, c=y, marker=(5, 1), alpha=0.8, lw=2, facecolors="none")
```

```
plt.xlim(0,1)
plt.ylim(0,1)
```

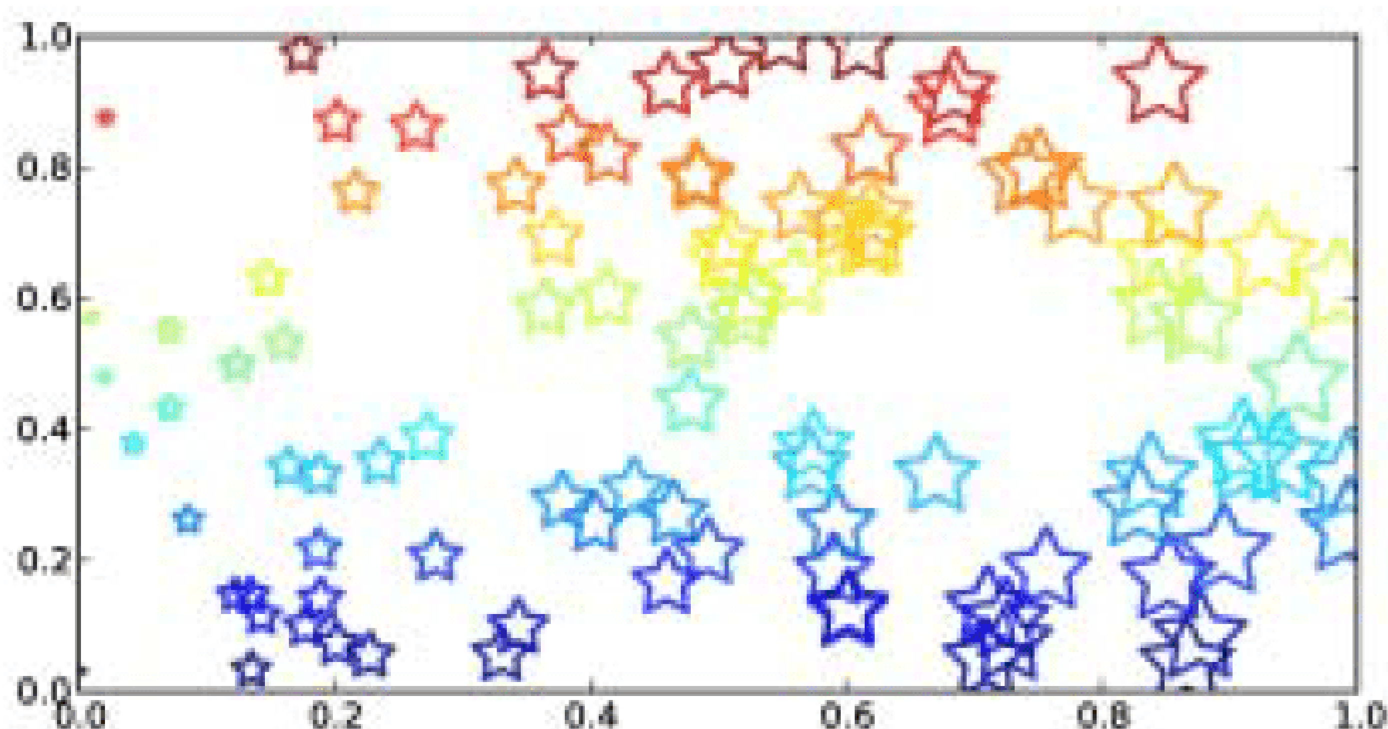


图 5-16　可指定每个点的颜色和大小的散列图

scatter()的前两个参数是数组，分别指定每个点的 X 轴和 Y 轴的坐标。s 参数指定点的大小，值和点的面积成正比。它可以是一个数，指定所有点的大小；也可以是数组，分别对每个点指定大小。

c 参数指定每个点的颜色，可以是数值或数组。这里使用一维数组为每个点指定了一个数值。通过颜色映射表，每个数值都会与一个颜色相对应。默认的颜色映射表中蓝色与最小值对应，红色与最大值对应。当 c 参数是形状为(N,3)或(N,4)的二维数组时，则直接表示每个点的 RGB 颜色。

marker 参数设置点的形状，可以是一个表示形状的字符串，也可以是表示多边形的两个元素的元组，第一个元素表示多边形的边数，第二个元素表示多边形的样式，取值范围为 0、1、2、3。0 表示多边形，1 表示星形，2 表示放射形，3 表示忽略边数而显示为圆形。

最后，通过 alpha 参数设置点的透明度，通过 lw 参数设置线宽，lw 是 line width 的缩写。facecolors 参数为"none"时，表示散列点没有填充色。

5.4.5　图像

imread()和 imshow()提供了简单的图像载入和显示功能。

imread()可以从图像文件读入数据，得到一个表示图像的 NumPy 数组。它的第一个参数是文件名或文件对象，format 参数指定图像类型，如果省略，就由文件的扩展名决定图像类型。对于灰度图像，它返回一个形状为(M,N)的数组；对于彩色图像，返回形状为(M,N,C)的数组。其中，M 为图像的高度，N 为图像的宽度，C 为 3 或 4，表示图像的通道数。下面的程序从“lena.jpg”中读入图像数据，得到的数组 img 是一个形状为(393, 512, 3)的单字节无符号整数数组。这是因为通常使用的图像都是采用单字节分别保存每个像素的红、绿、蓝三个通道的分量：

```
>>> img = plt.imread("lena.jpg")
```

```
>>> img.shape
>>> (393, 512, 3)
>>> img.dtype
dtype('uint8')
```

imshow()可以用来显示 imread()返回的数组。如果数组是表示多通道图像的三维数组，那么每个像素的颜色由各个通道的值决定：

```
>>> plt.imshow(img) # 注意图像是上下颠倒的
```

请注意，从 JPG 图像中读入的数据是上下颠倒的，为了正常显示图像，可以将数组的第 0 轴反转，或者设置 imshow()的 origin 参数为"lower"，从而让所显示图表的原点在左下角：

```
>>> plt.imshow(img[::-1]) # 反转图像数组的第 0 轴
>>> plt.imshow(img, origin="lower") # 让图表的原点在左下角
```

如果三维数组的元素类型为浮点数，那么元素的取值范围为 0.0 到 1.0，与颜色值 0 到 255 对应。超出这个范围可能会出现颜色异常的像素。下面的例子将数组 img 转换为浮点数组并用 imshow()进行显示：

```
>>> img = img[::-1]
>>> plt.imshow(img*1.0) # 取值范围为 0.0 到 255.0 的浮点数组，不能正确显示颜色
>>> plt.imshow(img/255.0) # 取值范围为 0.0 到 1.0 的浮点数组，能正确显示颜色
>>> plt.imshow(np.clip(img/200.0, 0, 1)) # 使用 clip()限制取值范围，整个图像变亮
```

如果 imshow()的参数是二维数组，就使用颜色映射表决定每个像素的颜色。下面显示图像中的红色通道：

```
>>> plt.imshow(img[:,:,0])
```

显示效果比较吓人，因为默认的图像映射将最小值映射为蓝色、将最大值映射为红色。我们可以使用 colorbar()将颜色映射表在图表中显示出来：

```
>>> plt.colorbar()
```

通过 imshow()的 cmap 参数可以修改显示图像时所采用的颜色映射表。颜色映射表是一个 ColorMap 对象，matplotlib 中已经预先定义好了很多颜色映射表，可以通过下面的语句找到这些颜色映射表的名字：

```
>>> import matplotlib.cm as cm
>>> cm._cmapnames
['Spectral', 'copper', 'RdYlGn', 'Set2', 'summer', 'spring', 'gist_ncar', ...]
```

下面使用名为 copper 的颜色映射表显示图像的红色通道，很有老照片的味道：

```
>>> plt.imshow(img[:,:,0], cmap=cm.copper)
```

读者可以运行下面的程序快速查看上面所有程序的显示效果，如图5-17所示(见文前彩插)。



matplotlib_imshow.py
用 imread()和 imshow()显示图像

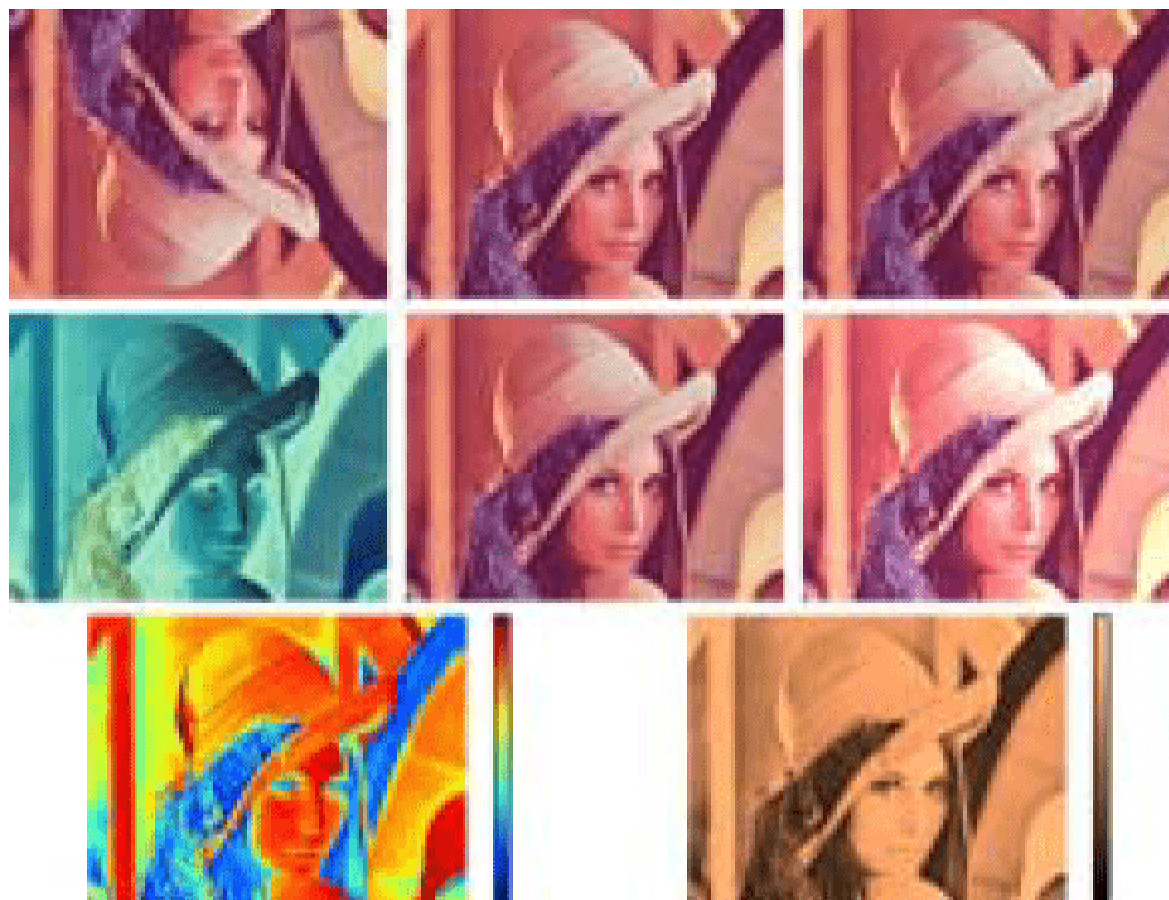
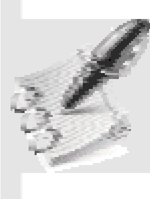


图 5-17　用 imread()和 imshow()显示图像

还可以使用 imshow()显示任意的二维数据，例如下面的程序使用图像直观地显示了二元函数 $f(x,y)=xe^{x^2-y^2}$，效果如图 5-18 所示(见文前彩插)。



matplotlib_2dfunc.py
使用 imshow()可视化二元函数

```
y, x = np.ogrid[-2:2:200j, -2:2:200j]
z = x * np.exp( - x**2 - y**2) ❶

extent = [np.min(x), np.max(x), np.min(y), np.max(y)] ❷

plt.figure(figsize=(10,3))
plt.subplot(121)
plt.imshow(z, extent=extent, origin="lower") ❷
plt.colorbar()
plt.subplot(122)
plt.imshow(z, extent=extent, cmap=cm.gray, origin="lower")
plt.colorbar()
```

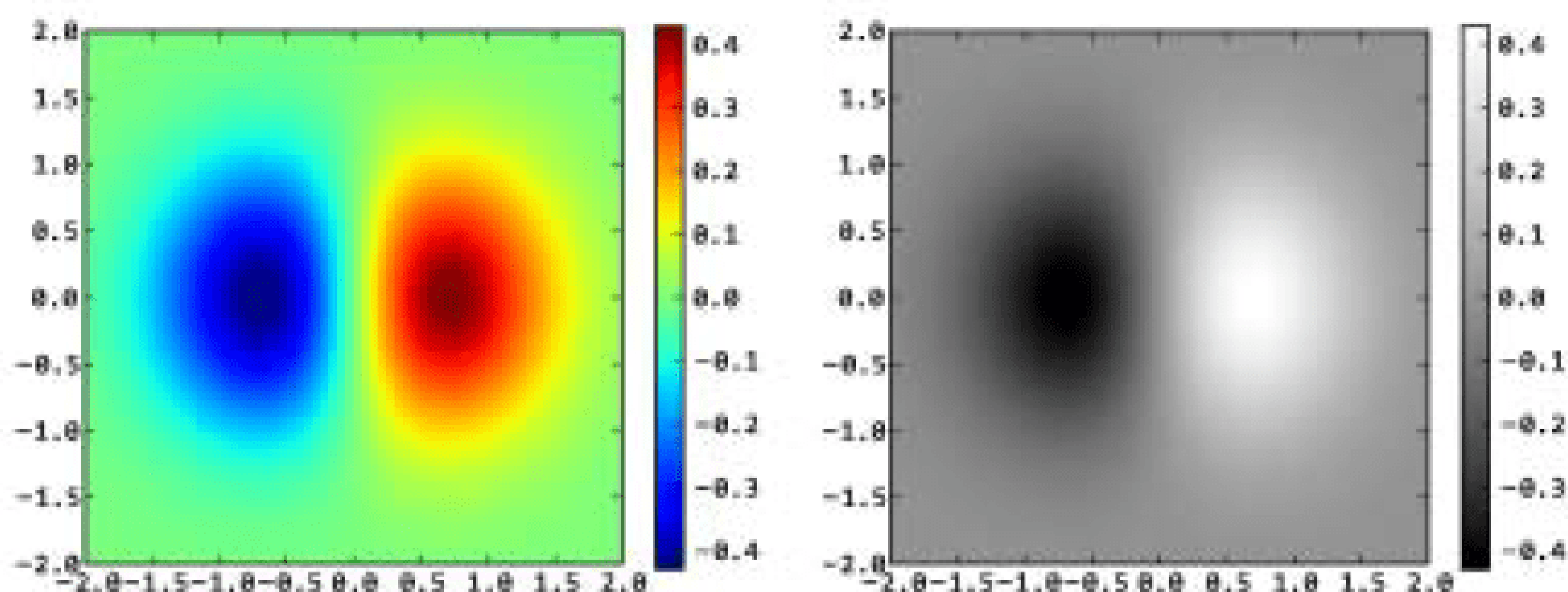


图 5-18　使用 imshow()可视化二元函数

❶首先通过数组的广播功能计算出表示函数值的二维数组 z，注意它的第 0 轴表示 Y 轴、第 1 轴表示 X 轴。❷然后将 X、Y 轴的取值范围保存到 extent 列表中。❸将 extent 列表传递给 imshow()的 extent 参数，这样一来，图表的 X、Y 轴的刻度标签将使用 extent 列表所指定的范围(见文前彩插)。

5.4.6　等值线图

还可以使用等值线图表示二元函数。所谓等值线，是指由函数值相等的各点连成的平滑曲线。等值线可以直观地表示二元函数值的变化趋势，例如等值线密集的地方表示函数值在此处的变化较大。matplotlib 中可以使用 contour()和 contourf()描绘等值线，它们的区别是：contourf()所得到的是带填充效果的等值线。下面的程序演示了这两个函数的用法，效果如图 5-19 所示(见文前彩插)：


matplotlib_contour.py
用 contour 和 contourf 描绘等值线图

```
y, x = np.ogrid[-2:2:200j, -3:3:300j] ❶
z = x * np.exp( - x**2 - y**2)

extent = [np.min(x), np.max(x), np.min(y), np.max(y)]

plt.figure(figsize=(10,4))
plt.subplot(121)
cs = plt.contour(z, 10, extent=extent) ❷
plt.clabel(cs) ❸
plt.subplot(122)
plt.contourf(x.reshape(-1), y.reshape(-1), z, 20) ❹
plt.show()
```

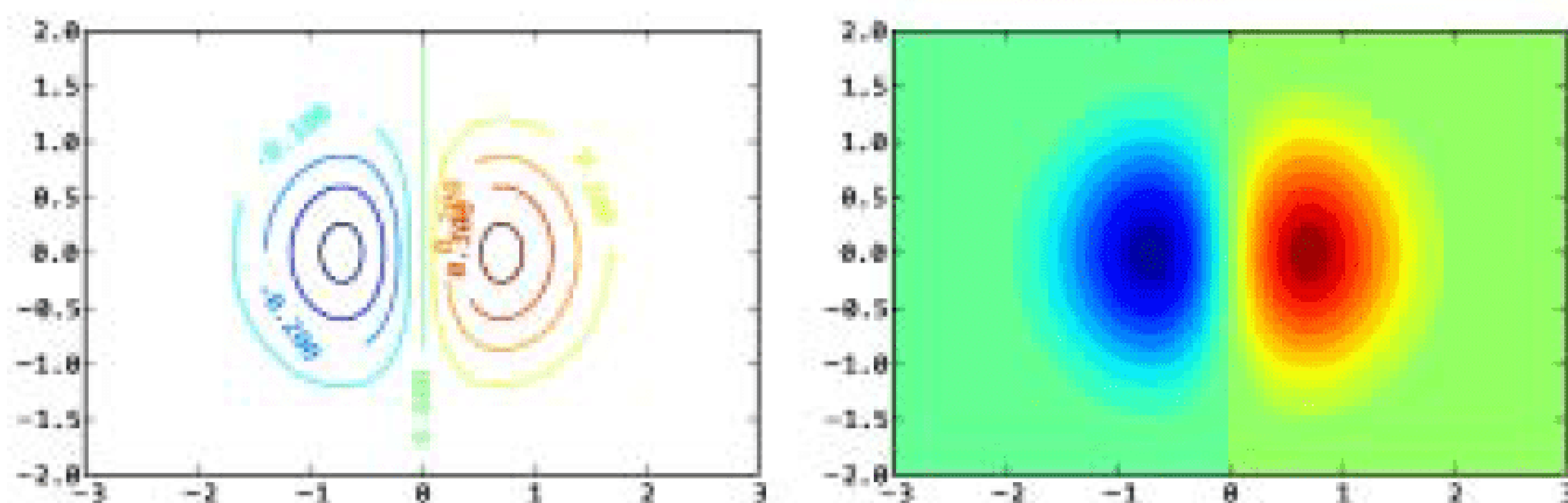


图 5-19　用 contour(左)和 contourf(右)描绘等值线图

❶为了更清楚地区分 X 轴和 Y 轴，这里让它们的取值范围和等分次数均不相同。这样得到的数组 z 的形状为(200, 300)，它的第 0 轴对应 Y 轴、第 1 轴对应 X 轴。

❷调用 contour()绘制数组 z 的等值线图，第二个参数为 10，表示将整个函数的取值范围等分为 10 个区间，即显示的等值线图中将有 9 条等值线。和 imshow()一样，可以使用 extent 参数指定等值线图的 X 轴和 Y 轴的数据范围。❸contour()所返回的是一个 QuadContourSet 对象，将它传递给 clabel()，为其中的等值线标上对应的值。

❹调用 contourf()，绘制将取值范围等分为 20 份、带填充效果的等值线图。这里我们演示了另外一种设置 X、Y 轴取值范围的方法。它的前两个参数分别是计算数组 z 时所使用的 X 轴和 Y 轴上的取样点，这两个数组必须是一维的。

我们还可以使用等值线绘制隐函数曲线。所谓隐函数，是指在一个方程 $f(x, y) = 0$ 中，若令 x 在某一区间内取任意值时总有相应的 y 满足此方程，则可以说方程 $f(x, y) = 0$ 在该区间上确定了 x 的隐函数 y，比如隐函数 $x^2 + y^2 - 1 = 0$ 表示一个单位圆。

显然，我们无法像绘制一般函数那样，先创建一个等差数组表示变量 x 的取值点，然后计算出数组中每个 x 所对应的 y 值。可以使用等值线解决这个问题，显然隐函数的曲线就是值等于 0 的那条等值线。下面的程序绘制函数 $f(x,y)=(x^2+y^2)^4-(x^2-y^2)^2$ 在 $f(x,y)=0$ 和 $f(x,y)-0.1=0$ 时的曲线，效果如图 5-20(左)所示(见文前彩插)。



matplotlib_implicit_func.py
使用等值线绘制隐函数曲线

```
y, x = np.ogrid[-1.5:1.5:200j, -1.5:1.5:200j]
f = (x**2 + y**2)**4 - (x**2 - y**2)**2

plt.figure(figsize=(9,4))
plt.subplot(121)
extent = [np.min(x), np.max(x), np.min(y), np.max(y)]
cs = plt.contour(f, extent=extent, levels=[0, 0.1],
    colors=["b", "r"], linestyles=["solid", "dashed"], linewidths=[2, 2])
```

在调用 contour()绘制等值线时，可以通过 levels 参数指定所绘制等值线对应的函数值，这里设置 levels 参数为[0, 0.1]，因此最终将绘制两条等值线。通过 colors、linestyles、linewidths 等参数可以分别指定每条等值线的颜色、线型以及线宽。

仔细观察图 5-20(左)会发现，表示隐函数 $f(x, y) = 0$ 蓝色实线并不是完全连续的，在图的中间部分它由许多孤立的小段构成。因为等值线在原点附近无限靠近，因此无论对函数 f 的取值空间如何进行细分，总是会有无法分开的地方，最终造成了图中的那些孤立的细小区域。而表示隐函数 $f(x, y) - 0.1 = 0$ 的红色虚线则是闭合且连续的。

可以通过 contour()返回的对象获得等值线上每点的数据，下面我们在 IPython 中观察变量 cs，它是一个 QuadContourSet 对象：

```
>>> run matplotlib_implicit_func.py
>>> cs
<matplotlib.contour.QuadContourSet instance at 0x0A340E90>
```

cs 对象的 collections 属性是一个等值线列表，每条等值线用一个 LineCollection 对象表示：

```
>>> cs.collections
<a list of 2 collections.LineCollection objects>
```

每个 LineCollection 对象都有它自己的颜色、线型、线宽等属性，注意这些属性所获得的结果外面还有一层封装，要获得其第 0 个元素才是真正的配置：

```
>>> c0.get_color()[0]
array([ 0.,  0.,  1.,  1.])
>>> c0.get_linewidth()[0]
2
```

由类名可知，LineCollection 对象是一组曲线的集合，因此它可以表示像蓝色实线那样由多条线构成的等值线。它的 get_paths()方法获得构成等值线的所有路径，本例中蓝色实线所表示的等值线由 42 条路径构成：

```
>>> len(cs.collections[0].get_paths())
42
```

路径是一个 Path 对象，通过它的 vertices 属性可以获得路径上所有点的坐标：

```
>>> path = cs.collections[0].get_paths()[0]
>>> type(path)
<class 'matplotlib.path.Path'>
>>> path.vertices
    array([[-0.08291457, -0.98938936],
           [-0.09039269, -0.98743719],
           [-0.09798995, -0.98513674],
```

```
        ...,
        [-0.05276382, -0.99548781],
        [-0.0678392 , -0.99273907],
        [-0.08291457, -0.98938936]])
```

介绍到这里，相信读者已经了解如何获得等值线上每点的数据了。下面的程序从等值线集合 cs 中找到表示等值线的路径，并使用 plot()将其绘制出来，效果如图 5-20(右)所示(见文前彩插)。

```
plt.subplot(122)
for c in cs.collections:
    data = c.get_paths()[0].vertices
    plt.plot(data[:,0], data[:,1],
        color=c.get_color()[0],  linewidth=c.get_linewidth()[0])
```

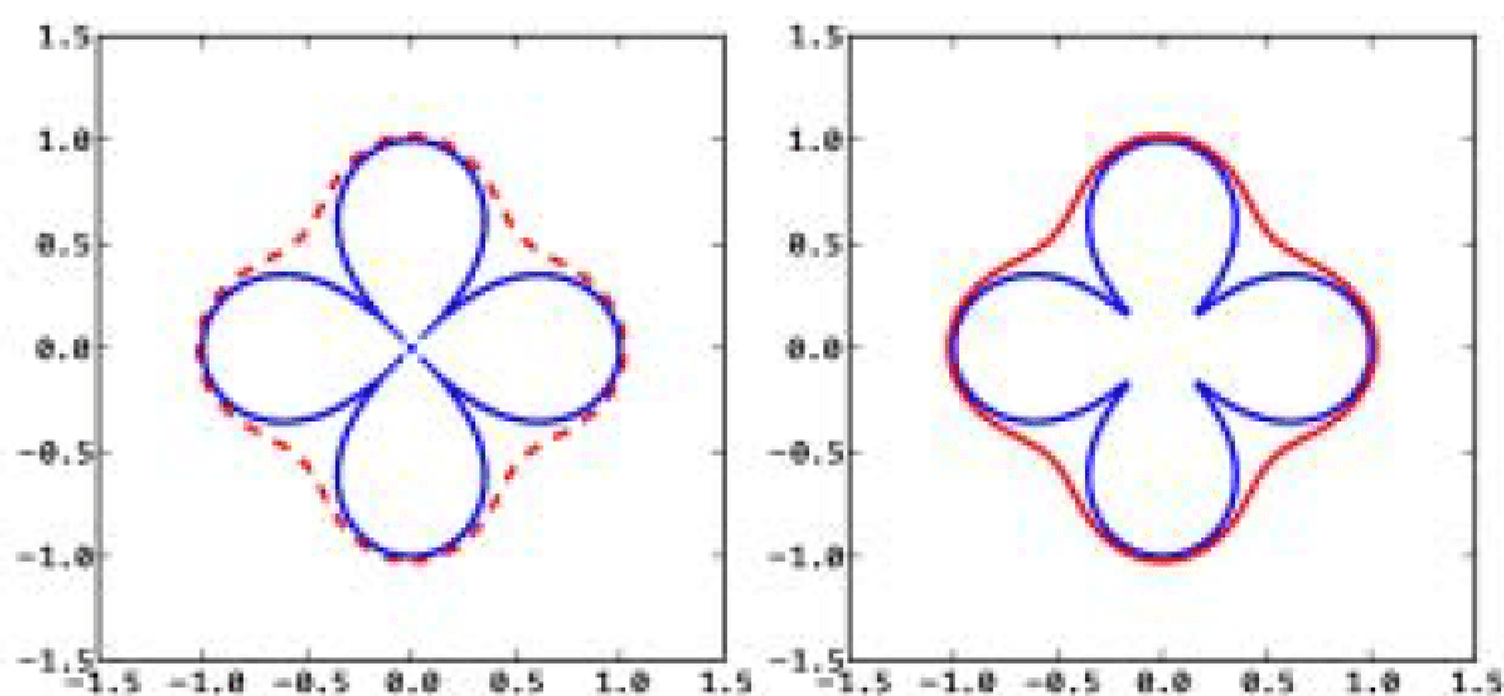


图 5-20　使用等值线绘制隐函数曲线(左)，获取等值线数据并绘图(右)

5.4.7　三维绘图

mpl_toolkits.mplot3d 模块在 matplotlib 基础上提供了三维绘图的功能。由于它使用 matplotlib 的二维绘图功能来实现三维图形的绘制工作，因此绘图速度有限，不适合用于大规模数据的三维绘图。如果读者需要更复杂的三维数据可视化功能，请阅读第 10 章。

下面是绘制三维曲面的程序：



matplotlib_surface.py
使用 matplotlib 绘制三维曲面

```
import numpy as np
import mpl_toolkits.mplot3d ❶
import matplotlib.pyplot as plt

x, y = np.mgrid[-2:2:20j, -2:2:20j] ❷
```

```
z = x * np.exp( - x**2 - y**2)

ax = plt.subplot(111, projection='3d') ❸
ax.plot_surface(x, y, z, rstride=2, cstride=1, cmap = plt.cm.Blues_r) ❹
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```

程序的输出如图 5-21 所示(见文前彩插)。

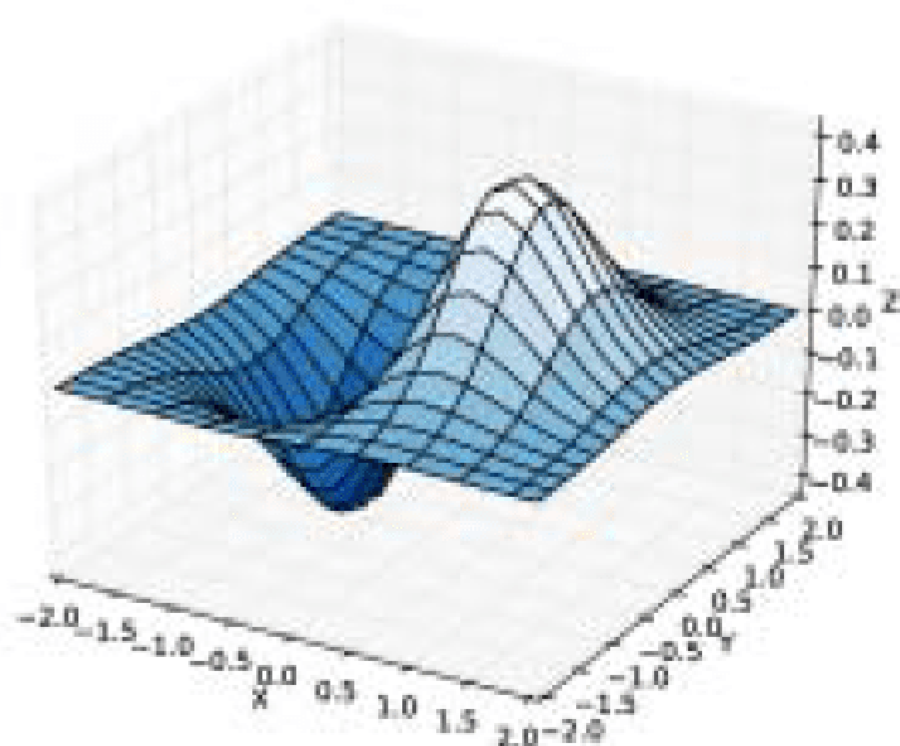


图 5-21　使用 mplot3D 绘制的三维曲面

❶首先载入 mplot3d 模块，matplotlib 中与三维绘图相关的功能均在此模块中定义。❷使用 mgrid 创建 X-Y 平面的网格并计算网格上每点的高度 z。由于绘制三维曲面的函数要求 X、Y 和 Z 轴的数据都用相同形状的二维数组表示，因此这里不能使用 ogrid 创建。和之前的 imshow() 不同，数组的第 0 轴可以表示 X 和 Y 轴中的任意一个，在本例中第 0 轴表示 X 轴、第 1 轴表示 Y 轴。

❸在当前图表中创建一个子图，通过 projection 参数指定子图的投影模式为"3d"，这样 subplot()将返回一个用于三维绘图的 Axes3D 子图对象。

投影模式

投影模式决定了点从数据坐标转换为屏幕坐标的方式。可以通过下面的语句获得当前有效的投影模式的名称:

```
>>> from matplotlib import projections
>>> projections.get_projection_names()
['3d', 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']
```

只有在载入mplot3d模块之后,此列表中才会出现'3d'投影模式。'aitoff'、'hammer'、'lambert'、'mollweide'等均为地图投影，'polar'为极坐标投影，'rectilinear'则是默认的直线投影模式。

❹调用 Axes3D 对象的 plot_surface()绘制三维曲面。其中：参数 x、y、z 都是形状为(20,20)的二维数组，数组 x 和 y 构成了 X-Y 平面上的网格，而数组 z 则是网格上各点在曲面上的取值。通过 cmap 参数来指定值和颜色之间的映射，即曲面上各点的高度值与其颜色的对应关系。rstride 和 cstride 参数分别是数组的第 0 轴和第 1 轴的下标间隔。对于很大的数组，使用较大的间隔可以提高曲面的绘制速度。程序中，plot_surface()调用和下面的语句是等价的：

```
ax.plot_surface(x[::2,:], y[::2,:], z[::2,:], rstride=1, cstride=1)
```

除了绘制三维曲面之外，Axes3D 对象还提供了许多其他的三维绘图方法。读者可以通过下面的链接地址找到各种三维绘图的演示程序：



http://matplotlib.sourceforge.net/examples/mplot3d/index.html
matplotlib 的三维绘图演示页面

第 6 章

Traits——为Python添加类型定义

Python 作为一种高级的动态编程语言，它的变量没有类型，这种灵活性给快速开发带来很多便利，不过也不是没有缺点。通过 Traits 库可以为对象的属性添加类型校验功能，从而提高程序的可读性，降低出错率。

Traits 库是由 Enthought 公司开发的一套开源扩展库。虽然 Traits 库本身和科学计算没有关系，但它却是该公司其他各种科学计算库的基础。因此我们将用一整章的篇幅对其进行详细介绍，为读者学习后续的各种科学计算扩展库打下基础。

6.1 开 发 背 景

Traits 库最初是为了开发交互式绘图库 Chaco 而设计的。通常绘图库中有很多表示图形的对象，每个对象都有很多诸如线型、颜色和字体之类的属性。为了方便用户使用，每个属性可以允许多种形式的值。例如颜色属性可以是'red'、0xff0000 或(255, 0, 0)。也就是说，可以用字符串、整数或元组等类型的数据表示颜色。这样的需求初看起来用 Python 的无类型属性是一个很好的选择，因为我们可以把各种各样的值赋给颜色属性。但是颜色属性虽然可以接受多样的值，却不是能接受所有的值，比如'abc'和 0.5 等等就不能很好地表示颜色。而且虽然为了方便用户使用，对外的接口可以接受多种类型的值，但是在程序内部必须有一个统一的表达方式以简化程序内部的实现。用 Trait 属性可以很好地解决这样的问题：

- 它可以接受能表示颜色的各种类型的值。
- 当给它赋值为不能表达颜色的值时，它能够立即捕捉到错误，并且提供一个有用的错误报告，告诉用户它能够接受什么样的值。
- 它提供一个内部的、标准的用于表示颜色的数据类型。

下面我们通过一个简单的实例演示 Trait 属性的功能。

```
from enthought.traits.api import HasTraits, Color ❶

class Circle(HasTraits): ❷
    color = Color ❸
```

❶首先载入 HasTraits 和 Color，建议读者在使用 Enthought 公司开发的扩展库时，采用和本例相同的载入方式。❷所有拥有 Trait 属性的类都需要从 HasTraits 继承。由于 Python 支持多

继承，因此很容易将现有的类改为支持 Trait 属性。❸Color 是一个 Trait 类型，在 Circle 类中用它定义了一个 color 属性。

熟悉 Python 的读者可能会觉得这个程序有些奇怪：按照标准的 Python 语法，直接在 class 下定义的属性 color 应该是 Circle 类的属性。而程序的目的是为 Circle 类的实例添加 color 属性。是不是应该在初始化方法__init__()中运行“self.color = Color”呢？答案是否定的，请记住：Trait 属性可以像类的属性那样来定义，像实例的属性那样来使用。我们不管 HasTraits 是如何实现这一点的，先看看如何使用 Trait 属性：

```
>>> c = Circle()
>>> Circle.color     #Circle 类没有 color 属性
Traceback (most recent call last):
AttributeError: type object 'Circle' has no attribute 'color'
>>> c.color
wx.Colour(255, 255, 255, 255)
```

从上面的运行结果可以看出：Circle 类没有 color 属性，而它的实例 c 却拥有 color 属性，默认值为白色“wx.Colour(255, 255, 255, 255)”，wx.Colour 是 wxPython 界面库所使用的颜色类型。

```
>>> c.color = "red"
>>> c.color
wx.Colour(255, 0, 0, 255)
>>> c.color = 0x00ff00
>>> c.color
wx.Colour(0, 255, 0, 255)
>>> c.color = (0, 255, 255)
>>> c.color
wx.Colour(0, 255, 255, 255)
>>> c.color = 0.5
[[省略]]
TraitError: The 'color' trait of a Circle instance must be a string of the form
(r,g,b) or (r,g,b,a) where r, g, b, and a are integers from 0 to 255, a wx.Colour
instance, an integer which in hex is of the form 0xRRGGBB, where RR is red, GG is
green, and BB is blue or 'aquamarine' or 'black' or 'blue violet' or 'blue' or
'brown' or 'cadet blue' or 'coral' or 'cornflower blue' or 'cyan' or [[此处略去很
多英文颜色名]] or 'yellow', but a value of 0.5 <type 'float'> was specified.
```

由上面的运行结果可知，可以将'red'、0x00ff00 和(0, 255, 255)等值赋给 color 属性，它们都被正确地转换为 wx.Colour 类型的值。而当赋值为 0.5 时，会抛出 TraitError 异常，并且显示一个很详细的出错信息来说明 color 属性能支持的所有值。最后看一个很酷的功能：

```
>>> c.configure_traits()
```

执行 configure_traits()之后，出现如图 6-1 所示的对话框以供我们修改颜色属性，任意选择一个颜色并单击 OK 按钮，configure_traits()返回 True，而 color 属性已经变为我们通过对话框所选择的颜色了：

```
>>> c.color
wx.Colour(64, 34, 117, 255)
```

如果在带“-wthread”参数的 IPython 中运行“c.configure_traits()”，会立即返回 False，而不会等待对话框关闭。

对于从 HasTraits 继承而来的对象，都可以调用其 configure_traits()方法以快速产生一个设置 Trait 属性的用户界面。在本例中，通过界面中的颜色输入框可以直接输入表示颜色的值，或者使用按钮打开颜色选择对话框。关于 Traits 库界面方面的功能将在下一章详细介绍。



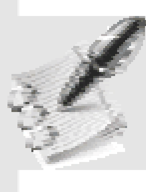
图 6-1　自动生成的用来修改颜色属性的对话框

6.2　Trait 属性的功能

Traits 库为对象的属性增加了类型定义的功能，此外还提供了如下的额外功能：

- 初始化：每个 Trait 属性都有自己的默认值。
- 验证：Trait 属性都有明确的类型定义，只有满足定义的值才能赋给属性。
- 代理：Trait 属性值可以代理给其他对象的属性。
- 监听：Trait 属性值发生变化时，可以运行事先指定的函数。
- 可视化：拥有 Trait 属性的对象可以很方便地生成编辑 Trait 属性的界面。

下面的例子展示了 Trait 属性的上述功能。



traits_demo.py
演示 Trait 属性的 5 项功能

```
class Parent ( HasTraits ):
    # 初始化: last_name 被初始化为'Zhang'
    last_name = Str( 'Zhang' ) ❶

class Child ( HasTraits ):
    age = Int

    # 验证: father 属性的值必须是 Parent 类的实例
    father = Instance( Parent ) ❷

    # 代理:  将 Child 对象的 last_name 属性代理给其 father 属性的 last_name
    last_name = Delegate( 'father' ) ❸

    # 监听: 当 age 属性的值被修改时，下面的函数将被运行
    def _age_changed ( self, old, new ): ❹
        print 'Age changed from %s to %s ' % ( old, new )
```

程序中定义了 Parent 和 Child 这两个从 HasTraits 继承而来的类，我们在 IPython 中载入这两个类，并分别创建它们的对象 p 和 c：

```
>>> from traits_demo import Parent, Child
>>> p = Parent()
>>> c = Child()
```

❶用 Str 类型定义 Parent 对象的 last_name 属性是一个字符串，并且它的默认值为'Zhang'：

```
>>> p.last_name
'Zhang'
```

❷用 Instance 类型定义 Child 对象的 father 属性是 Parent 类的实例，而 father 属性的默认值为 None。



如果 Parent 类在 Child 类之后定义，就可以用字符串表示类：father = Instance('Parent')。

❸通过 Delegate 类型，为 Child 对象创建了一个代理属性 last_name。代理功能将使得 c.last_name 和 c.father.last_name 始终拥有相同的值。但是由于还没有设置对象 c 的 father 属性，因此无法正确获得对象 c 的 last_name 属性：

```
>>> c.last_name
Traceback (most recent call last):
```

```
AttributeError: 'NoneType' object has no attribute 'last_name'
```

设置了对象 c 的 father 属性之后，就可以正确获取它的 last_name 属性了。并且对象 c 和 p 的 last_name 属性将始终保持一致：

```
>>> c.father = p
>>> c.last_name
'Zhang'
>>> p.last_name = "ZHANG"
>>> c.last_name
'ZHANG'
```

❹当对象 c 的 age 属性值发生变化时，将调用其监听函数 age_changed()：

```
>>> c.age = 4
Age changed from 0 to 4
```

最后，调用 configure_traits()显示一个修改属性值的对话框，如图 6-2(左)所示：

```
>>> c.configure_traits()
```



图 6-2　为 Child 对象自动生成的属性修改对话框(左)、单击 Father 按钮后弹出编辑 Parent 对象的对话框(右)

从自动生成的界面可以看到，属性按照其英文名排序，垂直排为一列。由于 father 属性是 Parent 类的对象，所以界面中以一个按钮表示它，单击此按钮将会弹出一个如图 6-2(右)所示的对话框，可以编辑 father 属性所对应的对象。

如果在编辑 Father 对象的对话框中修改 last_name 属性，Child 对象的 last_name 属性也同时被修改，这是因为 Child 对象的 last_name 属性是一个代理属性，值和 father.last_name 始终保持一致。

还可以调用 print_traits()，输出所有的 Trait 属性名和属性值：

```
>>> c.print_traits()
age:       4
father:    <__main__.Parent object at 0x13B49120>
last_name: u'Zhang'
```

或者调用 get()，得到一个描述对象所有 Trait 属性的字典：

```
>>> c.get()
{'age': 4, 'last_name': u'Zhang', 'father': <__main__.Parent object at 0x13B49120>}
```

此外还可以调用 set()来设置 Trait 属性的值，用 set()可以同时配置多个 Trait 属性：

```
>>> c.set(age = 6)
Age changed from 4 to 6
<__main__.Child object at 0x13B494B0>
```

在创建 HasTraits 的派生对象时可以使用关键字参数设置各个 Trait 属性的值，例如：

```
>>> c2 = Child(father=p, age=3)
```

当派生类中定义了__init__()时，在其中必须调用其父类的__init__()方法，否则 Trait 属性的一些功能将无效。

也许读者会对 Trait 属性的工作原理感兴趣，下面简单地介绍这些功能是如何实现的。

首先 Trait 属性本身和普通 Python 对象的属性是一样的。但是每个 Trait 属性都有一个 CTrait 对象与之对应，这个 CTrait 对象为 Trait 属性提供了许多额外的功能。可以通过 trait('属性名')获得与某个属性相对应的 CTrait 对象，或者用 traits()获得包含所有 CTrait 对象的字典。下面的语句获得 age 属性对应的 CTrait 对象：

```
>>> c.trait("age")
<enthought.traits.traits.CTrait object at 0x2A010A80>
```

Trait 属性的默认值保存在与其对应的 CTrait 对象中：

```
>>> p.trait("last_name").default
'Zhang'
```

给 Trait 属性赋值时的验证工作由 CTrait 对象的 validate()完成。当验证失败时抛出异常，验证成功时则返回所要赋的值。因此，validate()还可以在对值进行处理后，再赋值给属性。下面直接调用 father 属性所对应 CTrait 对象的 validate()：

```
>>> c.trait("father").validate(c, "father", 2)
TraitError: The 'father' trait of a Child instance must be a Parent or None,
but a value of 2 <type 'int'> was specified.
>>> c.trait("father").validate(c, "father", p)
<__main__.Parent object at 0x27DB7180>
```

当 Trait 属性值被改变时，HasTraits 对象的 trait_property_changed()[1]会被调用，在此方法中

① trait_property_changed()在 HasTraits 的父类 CHasTraits 中定义，CHasTraits 是用 C 语言实现的，可以在“ctraits.c”中找到它的源程序。

将会调用用户定义的属性监听函数。注意，它只调用监听函数，并不会修改属性的值，因此下面的语句将调用_age_changed()，但不会修改 age 属性的值：

```
>>> c.trait_property_changed("age", 8, 10)
Age changed from 8 to 10
>>> c.age # age 属性的值没有发生变化
6
```

CTrait 对象是连接 Trait 属性和 Trait 类型的纽带，通过 CTrait 对象的 trait_type 属性可以获得定义 Trait 属性时所使用的 Trait 类型：

```
>>> c.trait("age").trait_type
<enthought.traits.trait_types.Int object at 0x2BD1EFD0>
>>> c.trait("father").trait_type
<enthought.traits.trait_types.Instance object at 0x2BD103D0>
```

6.3 Trait 类型对象

在程序中使用 Trait 属性需要按照下面三个步骤进行：

(1) 从 enthought.traits.api 中载入所需要的对象。

(2) 创建 Trait 类型对象。

(3) 创建一个从 HasTraits 类继承的新类，在其中使用所创建的 Trait 类型对象定义 Trait 属性。

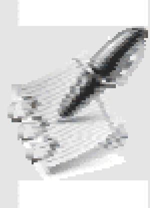
通常步骤(2)和步骤(3)是放在一起的，也就是说，在创建 Trait 类型对象的同时定义 Trait 属性，本书的大部分例子都是采用这种方式。例如：

```
class Person(HasTraits):
    age = Int(30)
    weight = Float
```

上面的程序为 Person 类定义了两个 Trait 属性：age 和 weight。其中：age 属性使用 Int 类型对象定义，30 为默认值；而 weight 属性则直接使用 Float 类型定义，实际上它也会创建一个 Float 类型对象，其具体实现在 HasTraits 类的内部进行。在 6.2 节我们介绍过，每个 Trait 属性都对应一个 CTrait 对象，而通过 CTrait 对象的 trait_type 属性可以获得 Trait 类型对象。实际上，这些 CTrait 对象和 Trait 类型对象都是在类中保存的，因此对于同一个 HasTraits 派生类的多个实例，它们的某个 Trait 属性所对应的 CTrait 对象都是同一个对象。下面创建两个 Person 类的实例，并分别查看它们的 Trait 属性所对应的 CTrait 对象和 Trait 类型对象，通过对象的地址可以看出多个实例之间共享 CTrait 对象的 Trait 类型对象。

```
>>> p1 = Person()
>>> p2 = Person()
>>> p1.trait("age")
<enthought.traits.traits.CTrait object at 0x03218500>
>>> p2.trait("age")
<enthought.traits.traits.CTrait object at 0x03218500>
>>> p1.trait("weight").trait_type
<enthought.traits.trait_types.Float object at 0x03319290>
>>> p2.trait("weight").trait_type
<enthought.traits.trait_types.Float object at 0x03319290>
```

也可以先单独创建一个 Trait 类型对象，然后用它定义多个 Trait 属性，下面是一个例子：



traits_create_type.py
重用 Trait 类型对象

```
from enthought.traits.api import HasTraits, Range

coefficient = Range(-1.0, 1.0, 0.0)

class Quadratic(HasTraits):
    c2 = coefficient
    c1 = coefficient
    c0 = coefficient
```

程序中，Quadratic 类有多个类型为 Range 的 Trait 属性，并且取值范围都是-1.0 到 1.0，初始值为 0.0。为了尽量重用代码，我们先创建了一个 Range 类型对象，然后使用它定义了三个 Trait 属性。为了进行比较，下面的代码直接在定义 Trait 属性时创建 Range 类型对象：

```
class Quadratic2(HasTraits):
    c2 = Range(-1.0, 1.0, 0.0)
    c1 = Range(-1.0, 1.0, 0.0)
    c0 = Range(-1.0, 1.0, 0.0)
```

Quadratic 对象的三个属性所对应的类型对象都是 coefficient，请注意比较 Range 类型对象的地址：

```
>>> run traits_create_type.py
>>> q = Quadratic()
>>> coefficient
<enthought.traits.trait_types.Range object at 0x06C27610>
>>> q.trait("c0").trait_type
<enthought.traits.trait_types.Range object at 0x06C27610>
```

```
>>> q.trait("c1").trait_type
<enthought.traits.trait_types.Range object at 0x06C27610>
```

而 Quadratic2 对象的属性所对应的类型对象是各不相同的：

```
>>> q2 = Quadratic2()
>>> q2.trait("c0").trait_type
<enthought.traits.trait_types.Range object at 0x06C54C30>
>>> q2.trait("c1").trait_type
<enthought.traits.trait_types.Range object at 0x06C54150>
```


6.4 Trait 的元数据

Trait 类型可以拥有元数据属性，这些属性保存在与 Trait 属性对应的 CTrait 对象中。下面以一个实例解释什么是元数据属性。



traits_metatest.py
测试 Trait 属性的元数据

```
from enthought.traits.api import HasTraits, Int, Str, Array, List

class MetadataTest(HasTraits):
    i = Int(99, myinfo="test my info") ❶
    s = Str("test", label=u"字符串") ❷
    # NumPy 的数组
    a = Array ❸
    # 元素为 Int 的列表
    list = List(Int)  ❹

test = MetadataTest()
```

在 IPython 中运行上面的程序之后，对 test 进行如下操作：

```
>>> test.traits()
{'i': <enthought.traits.traits.CTrait object at 0x05D44EA0>,
 's': <enthought.traits.traits.CTrait object at 0x05D44EF8>,
 'trait_added': <enthought.traits.traits.CTrait object at 0x06F90818>,
 'trait_modified': <enthought.traits.traits.CTrait object at 0x06F907C0>,
 [[省略]]}
>>> test.trait("i")
<enthought.traits.traits.CTrait object at 0x05D44EA0>
```

通过调用 HasTraits 对象的 traits()，可以得到一个包含其中所有 CTrait 对象的字典[②]。CTrait 对象用于描述 Trait 属性，例如 test.trait('i')描述 test.i、test.trait('s')描述 test.s。

每个 Trait 属性都有一个与之对应的 CTrait 对象用于描述它。所谓元数据属性，就是描述 Trait 属性的属性，它们保存在 CTrait 对象中。元数据属性可以分为三类：

- 内部属性：这些属性是 CTrait 对象自带的，只读不能写。
- 识别属性：这些属性可以自由设置，它们可以改变 Trait 属性的一些行为。
- 用户属性：用户自己添加的属性，需要自己编写程序使用它们。

下面是一些内部元数据属性，可以读取它们的值，但不能修改：

- array：是否是数组，不是数组的 Trait 属性没有此属性。
- default：Trait 属性的默认值。
- default_kind：一个描述默认值类型的字符串，可以是"value"、"list"、"dict"、"self"、"factory"、"method"等。
- trait_type：定义 Trait 属性时使用的 Trait 类型对象。
- inner_traits：内部的 CTrait 对象，在 List、Dict 等中使用，用来描述 List 和 Dict 的内部元素。
- type：Trait 属性的分类，可以是"constant"、"delegate"、"event"、"property"、"trait"。

下面的元数据属性不是预定义的，但是可以被 HasTraits 对象使用：

- desc：描述 Trait 属性用的字符串，在生成界面时使用它作为所创建的编辑器的帮助信息。
- editor：指定在界面中编辑 Trait 属性时所使用的编辑器类型。
- label：界面中 Trait 属性编辑器的标签字符串。
- rich_compare：指定判断 Trait 属性值发生变化的方式。默认值为 True，表示按值比较，False 表示按照对象地址比较。
- trait_value：指定 Trait 属性是否接受 TraitValue 类的对象，默认值为 False。为 True 时，将 Trait 属性设置为 TraitValue()，Trait 属性重置为默认值。
- transient：指定当对象被保存(持久化)时是否保存此 Trait 属性值，当此属性不存在时使用默认值 True。

下面查看一下上一实例中，test 对象的各个 Trait 属性的元数据属性。

❶在创建定义属性 i 的 Int 类型对象时，设置其默认值为 99，并设置了一个名为 myinfo 的用户元数据属性。这些信息都保存在与 i 属性对应的 CTrait 对象中：

```
>>> test.trait("i").default
99
>>> test.trait("i").myinfo
'test my info'
>>> test.trait("i").trait_type
```

② test 对象有两个额外的 CTrait 对象：trait_added 和 trait_modified，它们在 HasTraits 类中定义。

```
<enthought.traits.trait_types.Int object at 0x05DBD2D0>
```

❷属性 s 的默认值为"test"，并且它有一个识别元数据属性的 label。在生成界面时，使用它作为编辑器的标签。为了在界面中使用中文，需要使用 Unicode 字符串。

```
>>> test.trait("s").label
字符串
>>> test.configure_traits() # 显示界面，注意观察名为"字符串"的标签
```

❸Array 类型用于定义值为 NumPy 数组的 Trait 属性，因此属性 a 的元数据属性 array 为 True：

```
>>> test.trait("a").array
True
```

❹属性 list 是一个元素类型为整数的列表。通过 inner_traits 元数据属性可以获得与列表元素对应的 CTrait 属性：

```
>>> test.trait("list")
<enthought.traits.traits.CTrait object at 0x033FCF50>
>>> test.trait("list").trait_type
<enthought.traits.trait_types.List object at 0x01B92B30>
>>> test.trait("list").inner_traits # list 属性内部元素所对应的 CTrait 对象
(<enthought.traits.traits.CTrait object at 0x07005348>,)
>>> test.trait("list").inner_traits[0].trait_type # 内部元素所对应的 Trait 类型对象
<enthought.traits.trait_types.Int object at 0x06FF5710>
```

6.5　预定义的 Trait 类型

Traits 库为 Python 的许多数据类型提供了预定义的 Trait 类型。对于 Python 的每个简单数据类型来说，都有两种 Trait 类型与之对应：

- 强制 Trait 类型：当强制类型的 Trait 属性被赋值为类型不匹配的数据时，会抛出异常。
- 自动 Trait 类型：类型不匹配时会自动调用此类型对应的转换函数进行类型转换。

表 6-1 对这两种 Trait 类型进了总结。

表 6-1　强制 Trait 类型与自动 Trait 类型的比较

强 制 类 型	自 动 类 型	内置默认值	自动转换函数
Bool	CBool	False	bool()
Complex	CComplex	0+0j	complex()

(续表)

强 制 类 型	自 动 类 型	内置默认值	自动转换函数
Float	CFloat	0.0	float()
Int	CInt	0	int()
Long	CLong	0L	int()
Str	CStr	''	str()
Unicode	CUnicode	u''	unicode()

下面的例子比较这两种类型：

```
from enthought.traits.api import HasTraits, CFloat, Float

class Person(HasTraits):
    cweight = CFloat(50.0)
    weight = Float(50.0)
```

程序中用自动 Trait 类型 CFloat 定义了一个 cweight 属性，它可以接收能转换为数值的字符串"90"，而 weight 属性则使用强制 Trait 类型 Float 定义，将它赋值为"90"会抛出异常：

```
>>> p = Person()
>>> p.cweight = "90"
>>> p.cweight
90.0
>>> p.weight = "90"
TraitError [[省略]]
```

可以想象 CFloat 的内部处理过程：它先将传入的值用内部函数 float()进行类型转换，然后再把结果赋值给 Trait 属性。

除了简单类型以外，Traits 库还定义了许多其他的常用数据类型。下面列出了一些常用的预定义 Trait 类型：

- Any：任何对象。

```
Any( [value = None, **metadata] )
```

- Array：NumPy 的数组。

```
Array( [dtype = None, shape = None, value = None, typecode = None, **metadata] )
```

- Button：按钮类型，通常用于触发事件，参数用于描述界面中按钮的样式。

```
Button( [label ="", image = None, style = "button", orientation = "vertical",
    width_padding = 7, height_padding = 5, **metadata] )
```

- Callable：可调用对象。

```
Callable( [value = None, **metadata] )
```

- CArray：可自动转换类型的 NumPy 数组，参数和 Array 相同。
- Class：Python 老式类的对象。

```
Class( [value, **metadata] )
```

- Code：某种编程语言的字符串。

```
Code( [value = "", minlen = 0, maxlen = sys.maxint, regex = "", **metadata] )
```

- Color：界面库中所采用的颜色对象。

```
Color( [*args, **metadata] )
```

- CSet：自动转换类型的集合对象。

```
CSet( [trait = None, value = None, items = True, **metadata] )
```

- Constant：常量对象，值不能改变，必须指定初始值。

```
Constant( value*[, ***metadata] )
```

- Dict：字典对象，为了方便使用，在 Traits 库中还预先定义了一些键的类型为字符串的字典类型，例如 DictStrAny、DictStrBool 等。

```
Dict([key_trait = None, value_trait = None, value = None, items = True, **metadata])
```

- Directory：表示某个目录的路径的字符串。

```
Directory( [value ="", auto_set = False, entries = 10, exists = False, **metadata] )
```

- Either：多个 Trait 类型的复合，例如 Either(Str, Float)表示定义的属性可以是字符串或浮点数。

```
Either( val1*[, *val2, ..., valN, **metadata] )
```

- Enum：枚举数据，值可以是候选值中的任意一个。

```
Enum( values*[, ***metadata] )
```

- Event：触发事件用的对象。

```
Event( [trait = None, **metadata] )
```

- Expression：Python 的表达式对象。

```
Expression( [value ="0", **metadata] )
```

- File：表示文件路径的字符串。

```
File( [value = "", filter = None, auto_set = False, entries = 10, exists = False,
      **metadata ] )
```

- Font：界面库中表示字体的对象。

```
Font( [*args, **metadata] )
```

读者可以查看各个 Trait 类型的文档以了解其具体用法。下面以枚举类型为例，介绍 Trait 类型的使用方法。使用 Enum 可以定义枚举类型，在 Enum 的定义中给出所有的候选值，这些值必须是 Python 的简单数据类型，例如字符串、整数、浮点数等等，候选值的类型可以不一样。可以直接将候选值作为参数，也可以将其放在列表中，第一个值为默认值：

```
class Items(HasTraits):
    count = Enum(None, 0, 1, 2, 3, "many")
    # 或者：
    # count = Enum([None, 0, 1, 2, 3, "many"])
```

下面是运行结果：

```
>>> item = Items()
>>> item.count = 2
>>> item.count = "many"
>>> item.count = 5
Traceback (most recent call last): [[省略]]
```

如果希望候选值是动态的，可以用 values 参数指定候选值所对应的属性名：

```
class Items(HasTraits):
    count_list = List([None, 0, 1, 2, 3, "many"])
    count = Enum(values="count_list")
```

在上面的 Items 类中，首先用 List 定义了一个列表类型的 count_list 属性，并为其指定了默认值。然后在用 Enum 定义枚举类型的属性时，使用 values 参数指定枚举属性的候选值为 count_list 属性中的元素。

```
>>> item = Items()
>>> item.count = 5
Traceback (most recent call last)
[[略去错误提示，此错误提示无法显示候选值列表]]
```

```
>>> item.count_list.append(5)
>>> item.count = 5     #由于候选值列表中有 5，因此赋值成功
>>> item.count
5
```

在上面的例子中，因为 5 不在 count_list 属性中，所以第一次将 count 属性赋值为 5 时会抛出异常。当将 5 添加到 count_list 属性中之后，就可以将 count 属性设置为 5 了。

6.6 Property 属性

在标准的 Python 语法中，可以使用 property()为类创建 Property 属性。Property 属性的用法和一般属性相同，但是在获取它的值或者给它赋值时会调用相应的方法。Traits 库也提供了类似的功能，但是其用法比标准 Python 的简单。我们先看一个例子：



traits_property.py
演示 Property 类型

```
from enthought.traits.api import HasTraits, Float, Property, cached_property

class Rectangle(HasTraits):
    width = Float(1.0)
    height = Float(2.0)

    #area 是一个属性，当 width、height 的值发生变化时，对应的_get_area 函数将被调用
    area = Property(depends_on=['width', 'height'])  ❶

    # 通过 cached_property 修饰器缓存_get_area()的输出
    @cached_property  ❷
    def _get_area(self): ❸
        "area 的 get 函数，注意此函数名和对应 Property 属性名的关系"
        print 'recalculating'
        return self.width * self.height
```

❶在 Rectangle 类中，使用 Property()定义了一个 area 属性。Traits 库的 Property 类型和 Python 的不同，它根据属性名直接决定属性所对应的方法。当读取 area 属性值时，得到的是❸_get_area()的返回值；而当设置 area 属性时，所设置的值将传递给_set_area()。由于在本例中没有定义_set_area()，因此 area 属性是只读的。此外，通过 depends_on 参数可以指定 Property 属性的依赖关系。本例中，当 Rectangle 对象的 width 和 height 属性值发生变化时，需要重新计算 area 属性。

❷_get_area()用@cached_property 进行修饰，这样_get_area()的返回值将被缓存，除非 area 属性所依赖的 width 和 height 属性值发生变化，否则将一直使用缓存值，而不会每次都调用

_get_area()。下面看看实际的运行效果：

```
>>> from traits_property import Rectangle
>>> r = Rectangle()
>>> r.area  # 第一次取得 area 时，需要进行计算
recalculating
2.0
>>> r.width = 10
>>> r.area # 修改 width 之后，取得 area，也需要进行计算
recalculating
20.0
>>> r.area # width 和 height 都没有发生变化，因此直接返回缓存值，没有重新计算
20.0
```

我们看到：通过 depends_on 和@cached_property，系统可以跟踪 area 属性的状态，判断是否需要调用_get_area()以重新计算 area 属性值。注意：在运行"r.width=10"之后，并没有立即调用_get_area()，它只是保存一个需要重新计算的标志，等到真正需要获取 area 的值时，_get_area()才会被调用。

如果显示出编辑对象 r 的界面，就可以看到属性依赖关系的强大之处。为了更加有趣一些，这里连续调用两次 edit_traits()，弹出图 6-3 所示的两个编辑界面：

```
>>> r.edit_traits()
<enthought.traits.ui.ui.UI object at 0x02FCD420>
>>> r.edit_traits()
<enthought.traits.ui.ui.UI object at 0x02FD68A0>
```



这里使用 edit_traits()生成界面，它和 configure_traits()的区别将在下一章介绍。

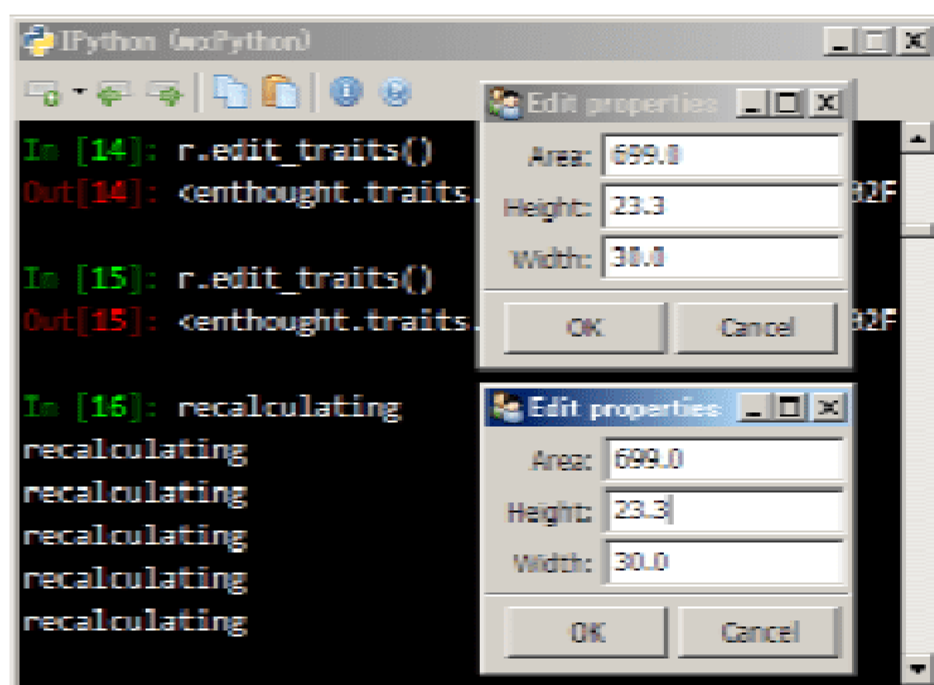


图 6-3　修改两个对话框中的 Height 或 Width 属性都会重新计算 Area，并同时更新对话框显示

修改任意一个界面中的 width 或 height 属性，在输入数值的同时，两个界面中的 Area、Height 和 Width 文本框中的值将同时更新，每次键盘按键都会调用_get_area()。此时在 IPython 窗口中

直接修改 width 的值，也会调用_get_area()：

```
>>> r.width = 25
recalculating
```

打开界面之后，界面对象开始监听对象 r 的各个属性，因此当我们修改 r.width 之后，系统设置 r.area 的标志为需要重新计算，然后发现有对象在监听 r.area 的值，因此直接调用_get_area()更新其值，并通知所有的监听对象，因此界面就一齐更新了。每个界面都会在 Trait 属性所对应的 CTrait 对象中添加监听对象：

```
>>> t = r.trait("area") #获得与 area 属性对应的 CTrait 对象
>>> t._notifiers(True) # _notifiers 方法返回所有的通知对象，当 aera 属性改变时，这些对象将
                        被通知
[<enthought.traits.trait_notifiers.FastUITraitChangeNotifyWrapper instance at ...>,
 <enthought.traits.trait_notifiers.FastUITraitChangeNotifyWrapper instance at ...>]
```

由于弹出了两个界面，因此有两个需要通知的对象。如果再运行一次 r.edit_traits()，这个列表将有 3 个元素。

6.7　Trait 属性监听

HasTraits 对象的所有 Trait 属性都自动支持监听功能。当某个 Trait 属性的值发生变化时，HasTraits 对象会通知所有监听此属性的函数。监听函数分为静态和动态两种，下面的程序演示了这两种监听方式：



traits_listener.py
演示 Trait 属性的监听功能

```
from enthought.traits.api import HasTraits, Str, Int

class Child ( HasTraits ):
    name = Str
    age = Int
    doing = Str

    def __str__(self):
        return "%s<%x>" % (self.name, id(self))

    # 当 age 属性的值被修改时，下面的函数将被运行
    def _age_changed ( self, old, new ): ❶
        print "%s.age changed: form %s to %s" % (self, old, new)
```

```
    def _anytrait_changed(self, name, old, new): ❷
        print "anytrait changed: %s.%s from %s to %s" % (self, name, old, new)

def log_trait_changed(obj, name, old, new): ❸
    print "log: %s.%s changed from %s to %s" % (obj, name, old, new)

if __name__ == "__main__":
    h = Child(name = "HaiYue", age=4)
    k = Child(name = "KaiYu", age=1)
    h.on_trait_change(log_trait_changed, name="doing") ❹
```

❶当 Child 对象的 age 属性值发生变化时，对应的静态监听函数_age_changed()将被调用。❷_anytrait_changed()是一个特殊的静态监听函数，任何 Trait 属性的值发生变化都会调用此函数。

❹通过调用 h.on_trait_change()，动态地将❸普通函数 log_trait_changed()和对象 h 的 doing 属性联系起来。当 doing 属性改变时，log_trait_changed()将被调用。动态监听函数也可以是某个对象的方法。

在 IPython 中运行下面的语句，观察各个属性监听函数的调用情况：

```
>>> run traits_listener.py
anytrait changed: <201ba80>.age from 0 to 4
<201ba80>.age changed: form 0 to 4
anytrait changed: HaiYue<201ba80>.name from  to HaiYue
anytrait changed: <201bae0>.age from 0 to 1
<201bae0>.age changed: form 0 to 1
anytrait changed: KaiYu<201bae0>.name from  to KaiYu
```

然后分别改变 h 和 k 的属性：

```
>>> h.age = 5
anytrait changed: HaiYue<5d87e70>.age from 4 to 5
HaiYue<5d87e70>.age changed: form 4 to 5
>>> h.doing = "sleeping"
anytrait changed: HaiYue<5d87e70>.doing from  to sleeping
log: HaiYue<5d87e70>.doing changed from  to sleeping
>>> k.doing = "playing"
anytrait changed: KaiYu<5d874e0>.doing from  to playing
```

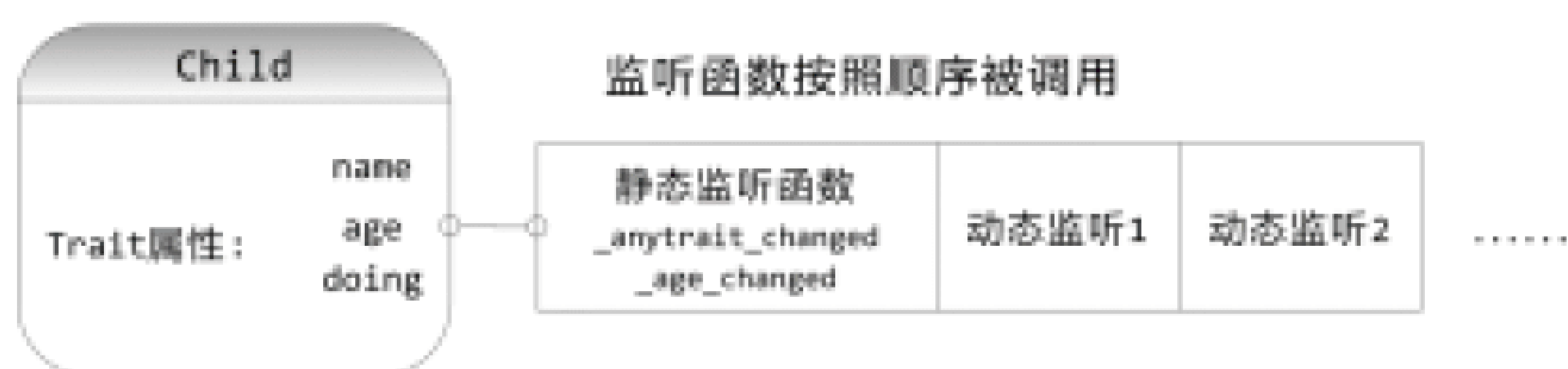


图 6-4　Trait 属性的监听函数的调用顺序

图 6-4 显示了各种监听函数的调用顺序。静态监听函数的参数有如下几种形式：

```
_age_changed(self)
_age_changed(self, new)
_age_changed(self, old, new)
_age_changed(self, name, old, new)
```

动态监听函数的参数有如下几种：

```
observer()
observer(new)
observer(name, new)
observer(obj, name, new)
observer(obj, name, old, new)
```

其中：obj 是拥有 Trait 属性的对象，name 为值发生变化的属性名，old 为改变之前的值，new 为改变之后的值。

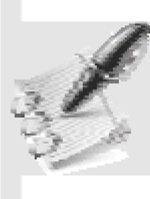
当多个 Trait 属性都需要使用同一个监听函数时，可以使用@on_trait_change 对监听函数进行修饰：

```
@on_trait_change( names )
def any_method_name( self, ...):
    ...
```

当 names 所描述的 Trait 属性发生改变时，将调用 any_method_name()，names 是一个字符串或列表，它能够很灵活地描述一组 Trait 属性。下面列举了一些常用的属性匹配语法，当与之匹配的属性发生变化时将调用被修饰的监听函数：

- 用逗号隔开多个属性名：'foo, bar' ，当 self.foo 或 self.bar 改变时。
- 用列表描述多个属性名：['foo','bar']，功能同上。
- 描述嵌套的属性名：'foo.bar'，当 self.foo.bar 或 self.foo 改变时。
- 描述嵌套的属性名：'foo:bar' ，当 self.foo.bar 改变时。
- 列表属性：'foo[]'，self.foo 是一个列表，当它本身或者它的元素改变时。
- 指定属性名开头字符串：'foo+'，当以 foo 开头的属性改变时。
- 指定元数据：'+foo'，有名为 foo 元数据的属性改变时。

完整的匹配方法请参考 Traits 库的用户手册。下面的程序演示了多种属性的匹配语法：



Traits_extended_name.py
演示属性名的匹配

```
from enthought.traits.api import *

class HasName(HasTraits):
    name = Str()

    def __str__(self):
        return "<%s %s>" % (self.__class__.__name__, self.name)

class Inner(HasName):
    x = Int
    y = Int

class Demo(HasName):
    x = Int
    y = Int
    z = Int(monitor=1) # 有元数据属性monitor的Int
    inner = Instance(Inner)
    alist = List(Int)
    test1 = Str()
    test2 = Str()

    def _inner_default(self):
        return Inner(name="inner1")

    @on_trait_change("x,y,inner.[x,y],test+,+monitor,alist[]")
    def event(self, obj, name, old, new):
        print obj, name, old, new

d = Demo(name="demo")
```

下面是各种属性值改变时的运行结果：

```
>>> d.x = 10 # 与x匹配
<Demo demo> x 0 10
>>> d.y = 20 # 与y匹配
<Demo demo> y 0 20
>>> d.inner.x = 1 # 与inner.[x,y]匹配
<Inner inner1> x 0 1
>>> d.inner.y = 2 # 与inner.[x,y]匹配
<Inner inner1> y 0 2
>>> d.inner = Inner(name="inner2") # 与inner.[x,y]匹配
<Demo demo> inner <Inner inner1> <Inner inner2>
>>> d.test1 = "ok" #与 test+匹配
<Demo demo> test1  ok
>>> d.test2 = "hello" #与test+匹配
<Demo demo> test2  hello
```

```
>>> d.z = 30  # 与+monitor匹配
<Demo demo> z 0 30
>>> d.alist = [3] # 与alist[]匹配
<Demo demo> alist [] [3]
>>> d.alist.extend([4,5]) #与alist[]匹配
<Demo demo> alist_items [] [4, 5]
>>> d.alist[2] = 10 # 与alist[]匹配
<Demo demo> alist_items [5] [10]
```

6.8　Event 和 Button 属性

Event 和 Button 是两个专门用以处理事件的 Trait 类型，Button 从 Event 继承，它除了具备 Event 的事件处理功能之外，还可以通过 TraitsUI 库自动生成界面中的按钮控件。Event 属性和其他的 Trait 属性有如下区别：

- 当任何值赋值给 Event 属性时都将触发与其绑定的属性监听事件，而通常的 Trait 属性只有在其值发生改变时才触发事件。
- Event 属性不存储属性值，因此对其赋值只是起到触发事件的作用，而所赋的值将被忽略。试图获取 Event 属性值将抛出异常。由于 Event 属性所触发的事件不表示某个属性值的变化，因此它们所对应的静态监听函数名为_event_fired，而不是_event_changed，下面是使用 Event 属性的例子：



traits_event.py
使用 Event 属性

```
class Point(HasTraits): ❶
    x = Float(0.0)
    y = Float(0.0)
    updated = Event

    @on_trait_change( "x,y" )
    def pos_changed(self): ❷
        self.updated = True

    def _updated_fired(self): ❸
        self.redraw()

    def redraw(self): ❹
        print "redraw at %s, %s" % (self.x, self.y)
```

❶在 Point 类中定义了 x、y 和 updated 三个 Trait 属性。❷使用@on_trait_change 对 pos_

changed()方法进行修饰，当x或y属性被修改时pos_changed()会被调用。这里是通过设置updated属性来触发 updated 事件。❸在 updated 的事件处理方法_updated_fired()中调用 redraw()重新绘制，下面是修改各种属性值时的事件触发情况：

```
>>> from traits_event import *
>>> p = Point()
>>> p.x = 1
redraw at 1.0, 0.0
>>> p.y = 1
redraw at 1.0, 1.0
>>> p.x = 1 # 由于 x 的值已经为 1，因此不触发事件
>>> p.updated = True
redraw at 1.0, 1.0
>>> p.updated = 0 # 给 updated 赋予任何值都会触发事件
redraw at 1.0, 1.0
```

6.9 Trait 属性的从属关系

代理功能是 Trait 属性的一个高级特性，它能够在不同的对象的 Trait 属性之间建立从属关系。类的继承关系通常描述的是对象之间“是”(is-a)的关系，而从属关系描述对象之间的包含关系，即“有”(has-a)的关系。

在 Traits 库中可以建立两种从属关系：代理(delegation)和原型(prototyping)。

- 代理：在两个属性之间建立的代理关系，能确保这两个属性值始终保持一致，修改任何一个属性的值都将同时修改另外一个属性的值。
- 原型：和代理类似，但是在派生属性的值被直接修改之后，两个属性的值不再保持一致，但它们仍然使用相同的校验方式。

下面用一个例子说明代理类型和原型类型的功能：



traits_delegate_intro.py
演示代理和原型类型

```
class Point(HasTraits):
    x = Int
    y = Int

class Circle(HasTraits):
    center = Instance(Point)     ❶
    x = DelegatesTo("center")    ❷
```

```
        y = PrototypedFrom("center") ❸
        r = Int

    p = Point()
    c = Circle()
    c.center = p
```

程序中定义了两个类——Point 和 Circle。❶Circle 对象的 center 属性是一个 Point 对象，❷❸属性 x 和 y 分别使用代理类型 DelegatesTo 和原型类型 PrototypedFrom 定义，我们称这两个属性为代理属性。它们都代理给了 center 属性，我们称 center 属性为代理对象。如果 c 是一个 Circle 对象，那么可以通过 c.x 访问 c.center.x。

下面是 Trait 库中关于代理类型的定义：

```
class DelegatesTo(delegate, prefix='', listenable=True, **metadata)
```

DelegatesTo 的第一个参数是一个字符串，它是拥有此代理属性的对象的某个属性名。当以下三种情况发生时，代理属性的值将会被改变：

- 代理对象对应的属性值改变时，即当 c.center.x 改变时，c.x 会同时被改变。
- 代理对象本身改变时，即当 c.center 改变时，c.x 和 c.y 都会同时被改变。
- 直接修改代理属性时，代理对象对应的属性值也将被改变，即可以通过修改 c.x 来改变 c.center.x。

PrototypedFrom 和 DelegatesTo 类似，但是当 c.y 被直接修改之后，c.y 和 c.center.y 的值将不再保持一致。下面用前面的例子演示代理类型和原型类型的功能以及它们的区别：

```
>>> p.x = 10     #修改代理对象，代理属性会同时改变
>>> c.x
10
>>> p.y = 100     #修改代理对象，原型属性会同时改变
>>> c.y
100
>>> c.x = 20     #修改代理属性，代理对象会同时改变
>>> p.x
20
>>> c.y = 200 # 由于c.y是一个原型属性，因此不会改变p.y
>>> p.y
100
>>> p.y = 300 # 再次修改p.y，由于关联已经断开，因此c.y不会改变
>>> c.y
200
```

当 prefix 参数为空字符串时，表示代理所涉及的两个属性名相同。如果 prefix 是一个合法的变量名，它将作为代理对象的属性名使用，此外还可以使用星号指定前缀。listenable 参数为

True，表示代理属性能够响应前两种情况的属性值改变事件。下面再看一个例子：



traits_delegate.py
用 DelegatesTo 建立代理关系

```
class System(HasTraits):
    name = Str

class CPU(HasTraits):
    cpu_type = Str

class PC(HasTraits):
    os = Instance(System)
    cpu = Instance(CPU)

    cpu_type = DelegatesTo("cpu") ❶
    os_name = DelegatesTo("os", prefix="name") ❷

    def _os_name_changed(self):
        print "OS changed to", self.os_name
```

程序中，PC 类有两个代理属性，❶其中 cpu_type 属性的代理对象是 cpu，由于没有指定 prefix 参数，因此代理属性 PC.cpu_type 和 PC.cpu.cpu_type 对应。❷而 os_name 属性则通过 prefix 参数指定它对应的代理对象的属性名，即 PC.os_name 和 PC.os.name 对应。

```
>>> os = System(name="WindowsXP")
>>> cpu = CPU(cpu_type="Atom280")
>>> pc = PC(os=os, cpu=cpu)
>>> pc.cpu_type # 和 pc.cpu.cpu_type 相同
'Atom280'
>>> pc.os_name # 和 pc.os.name 相同
'WindowsXP'
>>> os.name = "Windows7" # 修改被代理的属性，触发代理属性的监听函数
OS changed to Windows7
```

6.10　动态添加 Trait 属性

前面介绍的都是在类的定义中声明 Trait 属性，在类的对象中使用 Trait 属性。由于 Python 是动态语言，因此 Traits 库提供了直接为某个特定的对象添加 Trait 属性的方法。

在下面的例子中，直接生成一个 HasTraits 类的实例 a，然后调用其 add_trait()方法动态地为 a 添加一个名为 x 的 Trait 属性，类型为 Float，初始值为 3.0：

```
>>> from enthought.traits.api import *
>>> a = HasTraits()
>>> a.add_trait("x", Float(3.0))
>>> a.x
3.0
```

接下来再创建一个 HasTraits 类的实例 b，用 add_trait()为 b 添加一个属性 a，指定其类型为 HasTraits 类的实例。然后把实例 a 赋值给实例 b 的属性 a：

```
>>> b = HasTraits()
>>> b.add_trait("a", Instance(HasTraits))
>>> b.a = a
```

然后为实例 b 添加一个类型为 Delegate 的属性 y，在 b.y 和 b.a.x 之间建立代理连接。modify=True 表示可以通过 b.y 修改 b.a.x 的值。我们看到，当 b.y 的值改为 10 时，a.x 的值也同时改变了：

```
>>> b.add_trait("y", Delegate("a", "x", modify=True))
>>> b.y
3.0
>>> b.y = 10
>>> a.x
10.0
```

实际上，通过赋值语句为 HasTraits 对象添加新属性时，这些属性都是 Trait 属性，只不过它们的类型为 Python，因此它们能够接收任何类型的对象，起不到校验的作用：

```
>>> class A(HasTraits):
...     pass
>>> a = A()
>>> a.x = 3
>>> a.y = "string"
>>> a.traits()
{...省略...
'x': <enthought.traits.traits.CTrait object at 0x01B4C660>,
'y': <enthought.traits.traits.CTrait object at 0x01B4C660>}
>>> a.trait("x").trait_type
<enthought.traits.trait_types.Python object at 0x01AF5B90>
>>> a.configure_traits() # 显示编辑对象 a 的界面
```

上面的程序将显示一个界面，可以编辑对象 a 的属性 x 和 y，通过控件修改属性值之后，我们会发现它们都变成了 unicode 类型。

6.11　创建自己的 Trait 类型

除了使用预定义的 Trait 类型之外，还可以使用下面三种方法创建新的 Trait 类型：

- 从 TraitType 类继承
- 使用 Trait()
- 定义 TraitHandler 类

6.11.1　从 TraitType 继承

几乎所有预定义的 Trait 类型都从 TraitType 类继承，可以通过继承 TraitType 或预定义的 Trait 类型来创建新的 Trait 类型。下面的程序演示了如何使用继承创建新的 Trait 类型：



traits_oddint.py
自定义奇数 Trait 类型

```
from enthought.traits.api import BaseInt

class OddInt( BaseInt ): ❶

    # 定义默认值
    default_value = 1 ❷

    # trait 类型的描述文字
    info_text = 'an odd integer'

    def validate( self, object, name, value ): ❸
        "校验值是否为奇数"
        value = super(OddInt, self).validate(object, name, value) ❹
        if (value % 2) == 1:
            return value

        self.error( object, name, value )
```

❶OddInt 类定义了一个奇数 Trait 类型，它从 BaseInt 而不是从 Int 继承。Int 和 BaseInt 完全相同，但是为了提高运算速度，Int 在 C 语言级别进行校验，它不会运行 Python 级别的 validate()方法。因此为了进行是否是奇数的校验，需要从 BaseInt 继承。

❷在 OddInt 类中定义了两个属性——default_value 和 info_text，它们覆盖了 BaseInt 中的值。请注意 OddInt 的基类中没有 HasTraits，因此这里定义的是类属性而不是 Trait 属性。

❸OddInt 作为 BaseInt 的派生类，可以重用或修改 BaseInt 中定义的各种方法。在 OddInt 的 validate()中，❹首先通过 super()调用 BaseInt 的 validate()，判断 value 是否是整数，然后再判断 value 是否是奇数。下面测试 OddInt 的奇数校验效果：

```
>>> class A(HasTraits):
...     v = OddInt
...
>>> a = A()
>>> a.v
1
>>> a.v = 3
>>> a.v = 2
Traceback (most recent call last):
TraitError: The 'v' trait of an A instance must be an odd integer, but ...
```

在 TraitType 的派生类中还可以定义 post_setattr()方法，此方法在设置 object 的 name 属性为 value 之后被调用：

```
post_setattr( self, object, name, value )
```

如果在派生类中定义了 get()或 set()，那么此 Trait 类型就变成了 Property 属性类型，validate()将不再起作用。get()的参数为：

```
get( self, object, name )
```

set()的参数为：

```
set( self, object, name, value )
```

当读取某对象的 Property 属性时，将调用与之对应的 Property 属性类型的 get()方法。对 Property 属性赋值时，则调用 Property 属性类型的 set()方法。下面的程序演示了如何创建 Property 属性类型：



traits_property_type.py
创建 Property 属性类型

```
from enthought.traits.api import TraitType, HasTraits, Float

class ScaledValue(TraitType):
    def init(self): ❶
        self.value = 0

    def get(self, object, name):
        print "get %s.%s" % (object, name)
        return self.value * object.scale ❷

    def set(self, object, name, value):
        print "set %s.%s = %s" % (object, name , value)
```

```
        self.value = value ❸

class A(HasTraits):
    scale = Float(1.0)
    t1 = ScaledValue ❹
```

在 ScaledValue 类中定义了 get()和 set()方法，因此它是一个 Property 属性类型。❶在 init()中初始化 value 属性为 0，init()在 TraitType 的__init__()的最后被调用。❷在 get()中，返回的是 Property 属性对象自己的 value 属性和拥有此 Property 属性的对象的 scale 属性的乘积。❸设置 Proerty 属性时将调用 set()，从而设置其 value 属性为指定的值。



由于 ScaledValue 是 Traits 类型，多个 A 对象的 t1 属性将公用同一个 Traits 类型对象，因此不能用此方法为每个对象的 t1 属性保存不同的 value 值。

❹A 类中的属性 t1 是 ScaledValue 类型，因此它的值将由给属性 t1 设置的值和 scale 属性的值决定。在下面的程序中，为对象 a 的属性赋值：属性 t1 赋值为 2，属性 scale 赋值为 3。因此最终获得的属性 t1 的值为它们的乘积 6：

```
>>> a = A()
>>> a.t1 = 2
set <__main__.A object at 0x276B0780>.t1 = 2
>>> a.t1
get <__main__.A object at 0x276B0780>.t1
2.0
>>> a.scale = 3
>>> a.t1
get <__main__.A object at 0x276B0780>.t1
6.0
```

6.11.2 使用 Trait()

使用 Trait()可以快速定义新的 Trait 类型，它的参数列表非常灵活，具体用法请参考用户手册或 Trait()的帮助文档。下面介绍几种常见的用法：

- Trait(1.0)：第一个参数指定默认值，类型由默认值的类型决定，相当于 Float(1.0)。
- Trait(0, 1, 2, "many")：枚举类型，第一个参数 0 为默认值。
- Trait([0, 1, 2, "many"])：也可以用列表定义枚举类型。
- Trait(MyClass)：值必须是 MyClass 或其子类的对象，默认值为 None，但不能被赋值为 None。
- Trait(None, MyClass)：同上，但可以赋值为 None。

- Trait(obj)：值必须是 MyClass 或其子类的对象，默认值为 obj(假设 obj 是 MyClass 的对象)。

当参数中有一个或多个字典对象时，定义的是一个映射类型，映射类型会为对象创建两个属性：

- 值为字典中某个键值的正常属性。
- **名为正常属性名**加下划线的影子属性，值为正常属性在字典中对应**的值**。

下面的例子定义了一个描述颜色的映射类型：

```
# 使用字典定义一个映射类型
black_color = Trait("black",
    {"black":0x000000, "white":0xffffff, "red":0xff0000})

# 定义一个默认值为"red"的映射类型
red_color = Trait("red", black_color)

class Shape(HasTraits):
    line_color = black_color
    fill_color = red_color
```

Shape 类有两个属性：line_color 和 fill_color。因为它们都是映射类型的属性，所以 Shape 对象还有两个影子属性：line_color_和 fill_color_。当 line_color 或 fill_color 改变时，对应的影子属性的值将自动根据定义类型的字典进行设置：

```
>>> s = Shape()
>>> s.line_color
'black'
>>> s.line_color_
0
>>> s.fill_color
'red'
>>> hex(s.fill_color_)
'0xff0000'
>>> s.fill_color = "white"
>>> hex(s.fill_color_)
'0xffffff'
>>> s.fill_color_ = 0 # 对影子属性赋值并不能改变其对应的正常属性值
>>> s.fill_color
'white'
```

在自己的类型中使用影子属性

影子属性可以在 TraitType 类的 post_setattr()中进行赋值，例如下面的程序将属性值的两倍赋值给影子属性：

```
class T(TraitType):
    def post_setattr( self, object, name, value ):
        object.__dict__[ name + '_' ] = value * 2
```

6.11.3 定义 TraitHandler 类

我们还可以将一个 TraitHandler 对象传递给 Trait()，为 Trait 类型提供校验功能。当对某个 Trait 属性赋值时，如果与此属性对应的 Trait 类型对象拥有一个 TraitHandler 对象，将调用此 TraitHandler 对象的 validate()方法对所赋的值进行校验，并且将 validate()的返回值赋值给 Trait 属性。下面的程序演示了 TraitHandler 的使用方法：

```
from enthought.traits.api import TraitHandler, HasTraits, Trait

class TraitEvenInteger(TraitHandler):
    def validate(self, object, name, value):
        if type(value) is int and value > 0:
            return value - value % 2
        self.error(object, name, value)

    def info(self):
        return 'a positive even integer'

class B(HasTraits):
    v = Trait(1, TraitEvenInteger())
```

TraitEvenInteger 从 TraitHandler 继承，validate()对值进行校验，info()返回一个描述用的字符串。当 value 为整数并且大于 0 时，validate()返回小于等于 value 的最大偶数，否则它抛出 TraitError 异常。

```
>>> b.v = 3
>>> b.v
2
>>> b.v = 8
>>> b.v
8
>>> b.v = -5
...
TraitError: The 'v' trait of a B instance must be a positive even integer,
but a value of -5 <type 'int'> was specified.
```

Traits 库中预定义了一些比较实用的 TraitHandler 派生类。例如，TraitPrefixList 接受所指定的字符串列表或字符串的唯一前缀，赋值给 Trait 属性的值是与前缀匹配的完整字符串：

```
>>> class Foo(HasTraits):
```

```
...     n = Trait("one",TraitPrefixList(["one","two","three"]))
...
>>> f = Foo()
>>> f.n = "th"
>>> f.n
'three'
>>> f.n = "o"
>>> f.n
'one'
>>> f.n = "t"
TraitError: The 'n' trait of a Foo instance must be 'one' or 'two' or 'three'
(or any unique prefix), but a value of 't' <type 'str'> was specified.
```

请读者参考 Traits 库的目录下的“trait_handlers.py”文件，了解更多的 TraitHandler 派生类及其用法。

第 7 章

TraitsUI——轻松制作用户界面

Python 有着丰富的界面开发库，除了默认安装的 Tkinter 以外，wxPython、pyQt4 等都是非常优秀的界面开发库。但是它们有一个共同的问题：需要开发者掌握众多的 API 函数，许多细节需要开发者自己进行配置，例如控件的属性、位置以及事件响应，等等。

在开发科学计算程序时，我们希望快速实现一个够用的界面，让用户能够交互式地处理数据，而又不希望在界面制作上花费过多的精力。以 Traits 库为基础、以 MVC 模式为设计思想的 TraitsUI 库就是实现这一理想的最佳方案。

MVC 模式

MVC 的英文全称为 Model-View-Controller，它的目的是实现一种动态的程序设计，简化程序的修改和扩展工作，并且使程序的各个部分能够充分地被重复利用。

Model(模型)：程序中存储数据以及对数据进行处理的部分。

View(视图)：程序的界面部分，实现数据的显示。

Controller(控制器)：起到视图和模型之间的组织作用，控制程序的流程，例如将界面的操作转换为对模型的处理。

7.1 默 认 界 面

TraitsUI 库是一套建立在 Traits 库基础之上的用户界面库。它和 Traits 库紧密相连，如果读者已经设计好了一个从 HasTraits 继承而来的类，那么直接调用其 configure_traits()方法，系统将会使用 TraitsUI 库自动生成对话框界面，以供用户交互式地修改对象的 Trait 属性。下面是一个简单的例子：



traitsUI_default_view.py
编辑 HasTraits 对象的对话框

```
from enthought.traits.api import HasTraits, Str, Int

class Employee(HasTraits):
    name = Str
```

```
    department = Str
    salary = Int
    bonus = Int

Employee().configure_traits()
```

此程序创建一个 Employee 类，然后调用 configure_traits()显示出如图 7-1(左)所示的默认界面。

在此自动生成的界面中，所有的属性都采用文本框进行编辑，并且每个文本框前面都有一个文字标签，上面的文字根据 Trait 属性名自动生成：第一个字母变为大写，所有的下划线变为空格。对话框的最下面提供了 OK 和 Cancel 按钮以便确定或取消对 Trait 属性的修改。

由于 salary 属性定义为 Int 类型，因此当输入不能转换为整数的字符串时，输入框将以红色背景提醒输入错误，并且 OK 按钮变成无效，如图 7-1(右)所示。

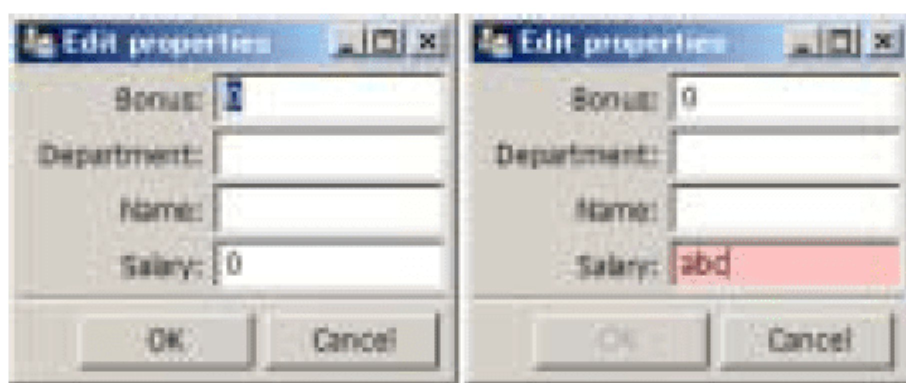


图 7-1 自动生成的 Employee 类的对话框(左)、提醒非法的输入数据并且使 OK 按钮无效(右)

没有写一行与界面相关的代码，就能得到一个已经够实用的界面，应该还是很令人满意的。如果想手工控制界面的设计和布局，就需要添加自己的代码了。

7.2 用 View 定义界面

HasTraits 的派生类用 Trait 属性保存数据，它相当于 MVC 模式中的模型(Model)。当没有指定界面的显示方式时，Traits 库会自动创建一个默认的界面。我们可以通过视图(View)对象为模型设计更加实用的界面。

7.2.1 外部视图和内部视图

下面是用视图对象定义界面的完整程序，图 7-2 显示的是界面截图。



traitsUI_simple_view.py
使用视图对象描述界面

```
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item ❶

class Employee(HasTraits):
```

```
    name = Str
    department = Str
    salary = Int
    bonus = Int

    view = View( ❷
        Item('department', label=u"部门", tooltip=u"在哪个部门干活"), ❸
        Item('name', label=u"姓名"),
        Item('salary', label=u"工资"),
        Item('bonus', label=u"奖金"),
        title = u"员工资料", width=250, height=150, resizable=True ❹
    )

if __name__ == "__main__":
    p = Employee()
    p.configure_traits()
```

此程序在模型类 Employee 的基础之上，添加了与界面显示相关的代码。❶界面相关的内容都位于 TraitsUI 库中，这里从中载入 View 和 Item。View 是描述界面的视图类，Item 是描述界面中的控件和模型对象的 Trait 属性之间关系的类。

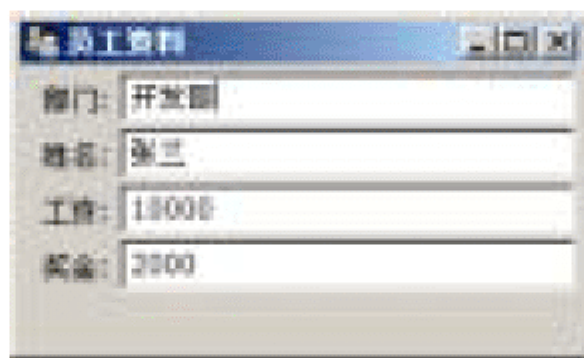


图 7-2　使用视图对象描述界面

❷在 Employee 类中创建了一个 View 对象，为 Employee 类的每个 Trait 属性创建对应的 Item 对象。❸创建多个 Item 对象，并作为参数传递给 View()。Item 对象是视图的基本组成单位，每个 Item 对象描述界面中的一个编辑器，这些编辑器用于编辑模型对象中对应的 Trait 属性的值。界面中的编辑器按照 Item 对象传递给 View()的先后顺序显示，而不再按照 Traits 属性名排序。Item 对象有许多参数，它们对 Item 对象的内容、表现以及行为进行描述。第一个参数指定与编辑器对应的 Trait 属性名，label 和 tooltip 参数设置编辑器的标签和提示文本。Item 对象还有很多其他属性，请读者参考 TraitsUI 的用户手册，或者在 IPython 中输入“Item??”直接查看其源代码。下面是 Item 类的部分源程序，Item 类从 HasTraits 继承，因此它的属性都是 Trait 属性：

```
class Item ( ViewSubElement ):
    """ An element in a Traits-based user interface.
    """

    #  Trait definitions:

    # A unique identifier for the item. If not set, it defaults to the value
    # of **name**.
    id = Str

    # User interface label for the item in the GUI. If this attribute is not
```

```
    # set, the label is the value of **name** with slight modifications:
    # underscores are replaced by spaces, and the first letter is capitalized.
    # If an item's **name** is not specified, its label is displayed as
    # static text, without any editor widget.
    label = Str

    # Name of the trait the item is editing:
    name = Str
```

除了 Item 之外，TraitsUI 模块还定义了 Item 的几个派生类：Label、Heading 和 Spring。它们只用于辅助界面布局，因此不需要和模型对象的 Trait 属性关联。

❹View 类也从 HasTraits 继承，可以直接在创建 View 对象时，通过关键字参数设置其 Trait 属性。title 属性为窗口标题栏中的文字，width 和 height 属性为窗口的大小，resizable 属性为 True 表示窗口的大小可变。

同一个模型对象可以通过多个视图对象用不同的界面显示，下面看一个例子：



traitsUI_views.py
使用多个视图对象

```
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Group, Item ❶

g1 = [Item('department', label=u"部门", tooltip=u"在哪个部门干活"), ❷
      Item('name', label=u"姓名")]
g2 = [Item('salary', label=u"工资"),
      Item('bonus', label=u"奖金")]

class Employee(HasTraits):
    name = Str
    department = Str
    salary = Int
    bonus = Int

    traits_view = View( ❸
        Group(*g1, label = u'个人信息', show_border = True),
        Group(*g2, label = u'收入', show_border = True),
        title = u"缺省内部视图")

    another_view = View( ❹
        Group(*g1, label = u'个人信息', show_border = True),
        Group(*g2, label = u'收入', show_border = True),
        title = u"另一个内部视图")

global_view = View( ❺
```

```
    Group(*g1, label = u'个人信息', show_border = True),
    Group(*g2, label = u'收入', show_border = True),
    title = u"外部视图")

p = Employee()

# 使用内部视图 traits_view
p.edit_traits() ❻

# 使用内部视图 another_view
p.edit_traits(view="another_view") ❼

# 使用外部视图 view1
p.configure_traits(view=global_view) ❽
```

❶从 TraitsUI 库载入 Group 类，用 Group 对象可以对界面中的编辑器进行分组。为了使后续定义视图对象的程序更加简洁，❷程序中定义了两个全局列表 g1 和 g2，它们的元素都是 Item 对象。❸❹在 Employee 类的内部用 View()定义了两个视图对象：traits_view 和 another_view。而❺定义了一个全局的视图对象：global_view。在定义视图对象时，用 Group 对用于定义界面上编辑器的 Item 对象进行分组。

值得注意的是，Employee 类中定义的两个视图对象既不是类的属性，也不是实例的属性。这些内部视图对象会放到 Employee 类的__view_traits__属性中。__view_traits__属性是一个 ViewElements 对象，它的 content 属性是保存所有内部视图的字典：

```
>>> run traitsUI_views.py
>>> Employee.__view_traits__.content.keys()
['another_view', 'traits_view']
```

❻当调用 edit_traits()显示界面时，默认使用在模型类内部定义的默认视图对象 traits_view 生成界面。❼使用 view 参数可以指定显示界面时所使用的内部视图对象的名称。❽也可以直接将视图对象传递给 view 参数，这样可以使用在模型类外定义的视图对象生成界面。图 7-3 显示了用三种视图对象生成的界面。

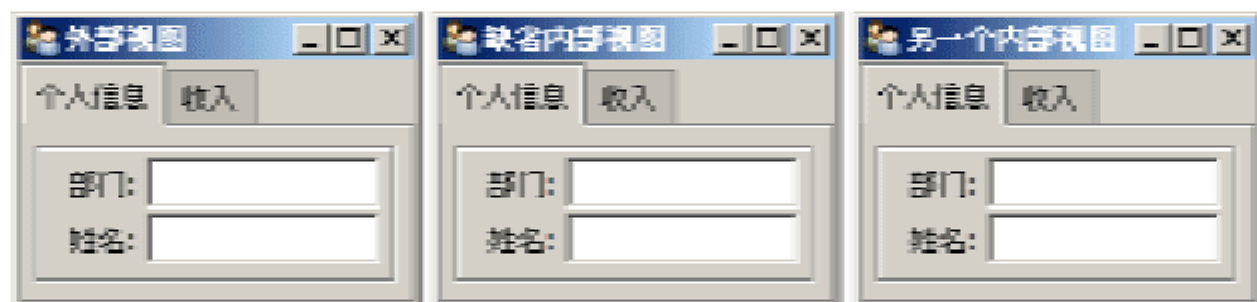


图 7-3　使用外部视图和内部视图定义界面的显示

edit_traits()和 configure_traits()一样，也被用于生成界面，它们的区别在于：edit_traits()显示界面之后不进入后台界面库的消息循环，因此如果直接运行只调用 edit_traits()的程序，界面将在显示之后立即关闭，程序的运行也随之结束。而对于 configure_traits()，将进入消息循环，

直到用户关闭所有窗口。因此通常情况下，主界面窗口或模态对话框[①]使用 configure_traits()显示，而无模态窗口或对话框则使用 edit_traits()显示。

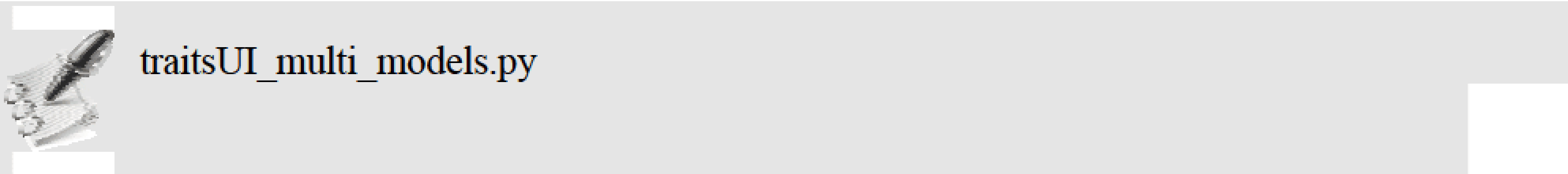
> **选择后台界面库**
>
> 用 TraitsUI 库创建的界面可以选择后台界面库，目前支持的有 qt4 和 wx 两种。在启动程序时添加“-toolkit qt4”或“-toolkit wx”可选择使用何种界面库生成界面。本书全部使用 wx 作为后台界面库。

在本小节的例子中，Employee 类用于保存数据，因此它属于 MVC 模式中的模型(Model)，而 View 对象定义了 Employee 的界面显示部分，它属于视图(View)。通过将视图对象传递给模型对象的 configure_traits()方法，将模型对象和视图对象联系起来。在用来定义编辑器的 Item 对象中，不直接引用模型对象的属性，而是通过属性名和模型对象进行联系。这样一来，模型和视图之间的耦合性很弱，只需要属性名匹配，同一个视图对象可以运用到不同的模型对象之上。

有时候我们希望模型类知道如何显示它自己，这时可以在模型类的内部定义视图。在模型类中定义的视图可以被其派生类继承，因此派生类能使用父类的视图。在调用 configure_traits()时如果不设置 view 参数，将使用模型对象内部的默认视图对象生成界面。如果在模型类中定义了多个视图对象，默认使用名为 traits_view 的视图对象。

7.2.2 多模型视图

在上一小节的例子中，一个模型可以对应多个视图。同样，使用一个视图可以将多个模型对象的数据显示在一个界面窗口中。下面是用一个视图对象同时显示多个模型对象的例子。程序的运行界面如图 7-4 所示。



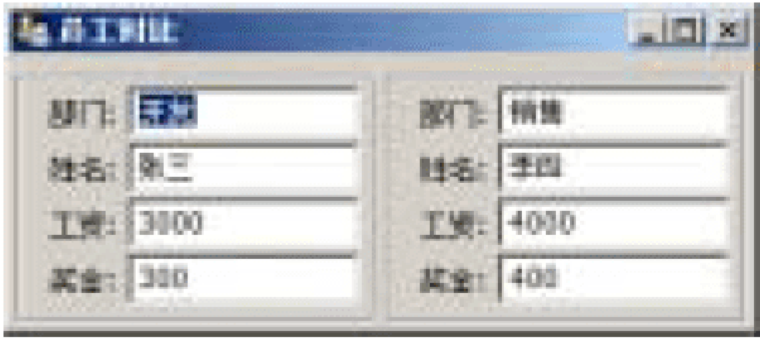


图 7-4 用一个视图对象同时显示多个模型对象

对于前面的模型类 Employee，我们可以设计如下的复合视图对象 comp_view，它能同时显示两个 Employee 对象：

```
comp_view = View(
```

① 在模态对话框显示时，程序的其他窗口均无法响应用户操作。

```
    Group(
        Group(
            Item('p1.department', label=u"部门"),
            Item('p1.name', label=u"姓名"),
            Item('p1.salary', label=u"工资"),
            Item('p1.bonus', label=u"奖金"),
            show_border=True
        ),
        Group(
            Item('p2.department', label=u"部门"),
            Item('p2.name', label=u"姓名"),
            Item('p2.salary', label=u"工资"),
            Item('p2.bonus', label=u"奖金"),
            show_border=True
        ),
        orientation = 'horizontal'
    ),
    title = u"员工对比"
)
```

注意：Item 对象的第一个参数不是简单的模型对象的属性名，它同时设置了 Item 对象的两个属性——object 和 name。例如，参数"p1.department"将设置 Item 对象的 object 属性为"p1"，name 属性为"department"。object 属性告诉 Item 对象如何找到模型对象，而 name 属性则告诉 Item 对象如何找到模型对象中与其对应的属性。

接下来，下面的程序生成组成模型对象的两个 Employee 对象——employee1 和 employee2：

```
employee1 = Employee(department = u"开发", name = u"张三", salary = 3000, bonus = 300)
employee2 = Employee(department = u"销售", name = u"李四", salary = 4000, bonus = 400)
```

在显示界面时，使用 context 参数将包含两个模型对象的字典传递给 configure_traits()：

```
HasTraits().configure_traits(view=comp_view, context={"p1":employee1, "p2":employee2})
```

通过 context 参数传递的实际上是视图对应的模型。这里的模型对象是一个字典，它的键和 Item 对象的 object 属性的值相同。由于已经通过 context 参数传递了模型对象，因此 configure_traits()方法原本所属的对象将不会被用作界面的模型对象。这里直接创建一个临时的 HasTraits 对象，然后调用其 configure_traits()方法。

如果读者认为这种写法有些取巧，可以直接调用视图对象的 ui()方法显示界面，它的参数就是界面所要显示的模型对象。由于 ui()和 edit_traits()一样[②] 不会开始界面库的消息循环，因此需要在运行 ui()之后添加开始消息循环的代码。下面的消息循环代码支持所有的后台界面库：

② 实际上，edit_traits()会调用视图对象的 ui()来显示界面。

```
comp_view.ui({"p1":employee1, "p2":employee2})

from enthought.pyface.api import GUI
GUI().start_event_loop() # 开始后台界面库的消息循环
```

如果读者只需要用 wx 库显示界面，也可以用下面的代码来运行 wx 库的消息循环：

```
import wx
wx.PySimpleApp().MainLoop() # 开始 wx 库的消息循环
```

7.2.3 Group 对象

在前面例子的视图定义中，我们通过 Group 对象将一组相关的 Item 对象组织在一起。本节详细介绍如何使用 Group 对象组织界面。

在程序“traitsUI_views.py”中，View 对象中直接放置了多个 Group 对象，效果如图 7-3 所示，它使用标签页的形式显示在 View 下定义的多个 Group 对象。

如果希望能同时看到两个 Group 对象，可以像程序“traitsUI_multi_models.py”一样，再创建一个 Group 对象，将那两个 Group 对象包括起来，效果如图 7-4 所示。由于外层 Group 对象的 orientation 属性为'horizontal'，因此它内部的两个 Group 对象将水平方向排列。下面的代码显示了 View 对象中 Group 对象的嵌套关系：

```
View(
    Group(
        Group(...),
        Group(...),
        orientation = 'horizontal'
    )
)
```

在创建 Group 对象时，可以通过设置其 orientation 和 layout 等属性，改变 Group 对象的内容呈现方式。由于某些设置会经常用到，因此 TraitsUI 还提供了几个 Group 的派生类，以覆盖这些属性的默认值。例如，下面是从 Group 类继承的 HSplit 类的代码：

```
class HSplit ( Group ):
    # ... ...
    layout      = 'split'
    orientation = 'horizontal'
```

HSplit 对象将其包括的内容按照水平方向排列，并且在两块区域之间添加一个可调整位置的分隔条。使用 HSplit 和如下的代码等价：

```
Group( ... , layout = 'split', orientation = 'horizontal')
```

为了正确显示分隔条，内容中需要有一个 Group 对象具有 scrollable 属性，如下面的省略代码所示：

```
view1 = View(
    HSplit(
        VGroup(
            ... ...,
            scrollable = True
        ),
        VGroup(
            ... ...
        ),
    ),
    resizable = True,
    width = 400,
    height = 150
)
```

下面的程序演示了 4 种不同的界面分组方式，效果如图 7-5 所示。



traitsUI_group.py
用 Group 对 View 中的编辑器进行分组

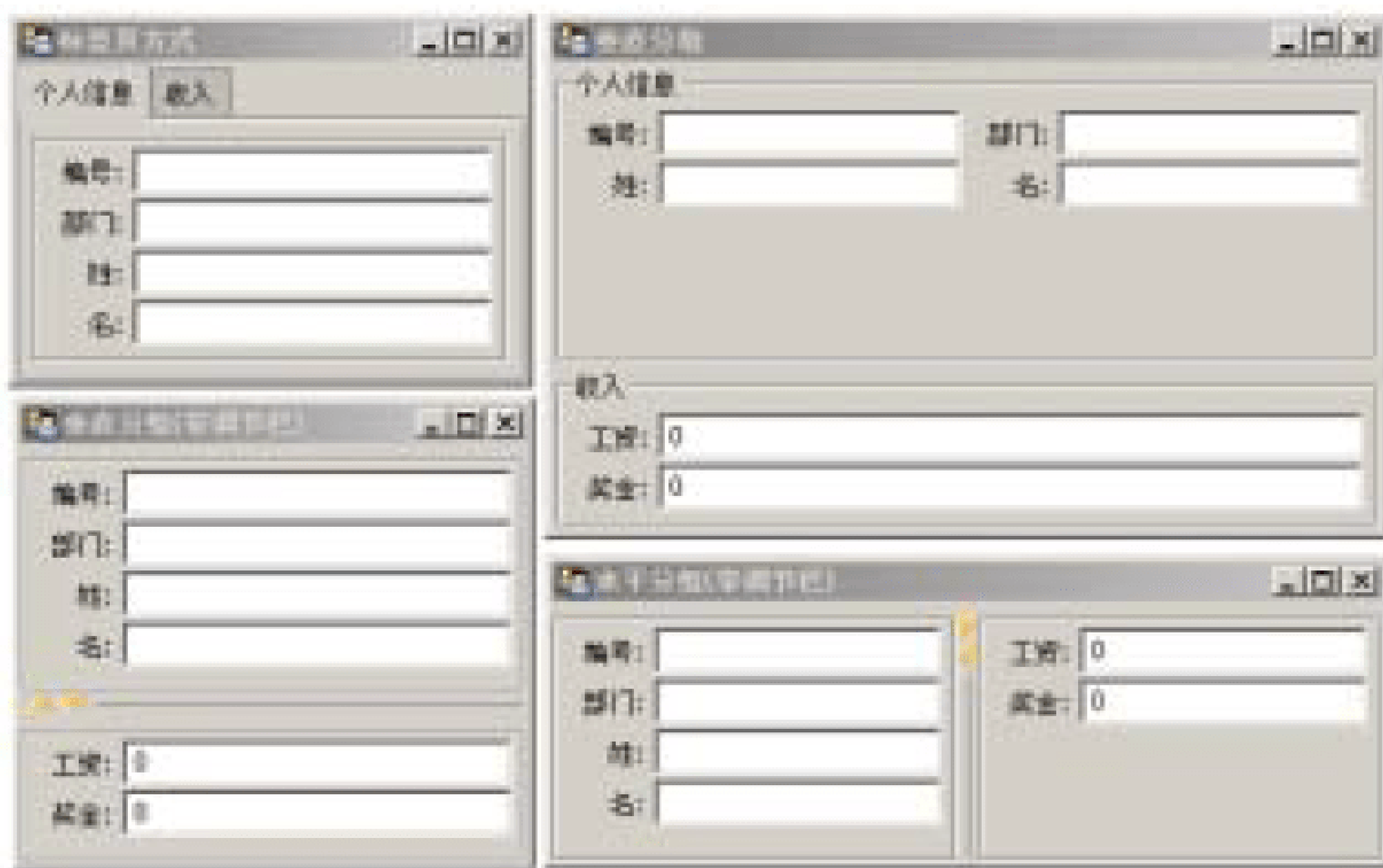


图 7-5　在界面中用 Group 对象进行分组

下面是 Group 的各种派生类及其对应的属性的默认设置：

- HGroup：内容水平排列。

```
Group(orientation= 'horizontal')
```

- HFlow：内容水平排列，当超过水平宽度时，将自动换行。show_labels 属性为 False，表示隐藏其中的所有编辑器的标签文字。

```
Group(orientation= 'horizontal', layout='flow', show_labels=False)
```

- HSplit：内容水平分隔，中间插入分隔条。

```
Group(orientation= 'horizontal', layout='split')
```

- Tabbed：内容分标签页显示。

```
Group(orientation= 'horizontal', layout='tabbed')
```

- VGroup：内容垂直排列。

```
Group(orientation= 'vertical')
```

- VFlow：内容垂直排列，当超过垂直高度时，将自动换列。

```
Group(orientation= 'vertical', layout='flow', show_labels=False)
```

- VFold：内容垂直排列，可折叠。

```
Group(orientation= 'vertical', layout='fold', show_labels=False)
```

- VGrid：按照多列网格进行垂直排列，columns 属性决定网格的列数。

```
Group(orientation= 'vertical', columns=2)
```

- VSplit：内容垂直排列，中间插入分隔条。

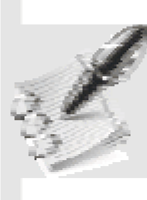
```
Group(orientation= 'vertical', layout='split')
```

除了上面的 orientation、layout、show_labels、columns 等属性之外，Group 对象还提供了许多其他的属性以控制其显示效果。和 Item 一样，读者可以在 Group 类的源程序中找到每个属性的详细说明。

下面我们通过一个例子说明 visible_when 和 enabled_when 属性的用法。这两个属性控制 Group 对象在界面中是否显示以及是否有效。



Item 也提供了 visible_when 和 enabled_when 属性，用法和 Group 的完全相同。



traitsUI_group_condition.py
控制 Group 的显示以及是否有效

```
class Shape(HasTraits):
    shape_type = Enum("rectangle", "circle")
    editable = Bool
    x, y, w, h, r = [Int]*5

    view = View(
        VGroup(
            HGroup(Item("shape_type"), Item("editable")),
            VGroup(Item("x"), Item("y"), Item("w"), Item("h"),
                visible_when="shape_type=='rectangle'", enabled_when="editable"),
            VGroup(Item("x"), Item("y"), Item("r"),
                visible_when="shape_type=='circle'",  enabled_when="editable"),
        ), resizable = True)
```

程序中，Shape 是一个表示矩形或圆形的类，其具体形状由 shape_type 属性决定，而图形的参数则由 x、y、w、h、r 等属性决定。editable 属性决定是否能通过用户界面修改图形参数。在视图定义中，使用 VGroup 对象定义了两个编辑器组，分别编辑矩形参数和圆形参数。通过设置 VGroup 的 visible_when 和 enabled_when 属性，将模型对象的 shape_type 属性和 editable 属性与编辑器的界面显示联系起来。

visible_when 和 enabled_when 属性都是表示布尔表达式的字符串。当布尔表达式中涉及的模型对象的属性发生变化时，将会对字符串进行求值，并根据求值结果更新界面的显示。图 7-6 是程序的显示效果，其中左图对应的 shape_type 属性为"rectangle"、editable 属性为 True。当 shape_type 属性为"rectangle"时，将显示矩形参数的编辑器而隐藏圆形参数的编辑器。当 editable 属性为 False 时，所有编辑器都变成无效。

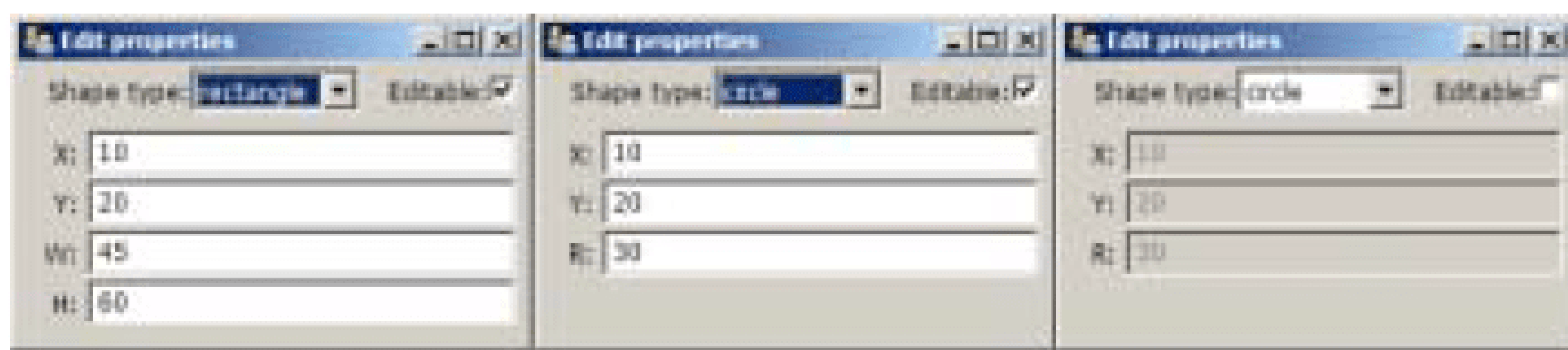


图 7-6　演示 visible_when 和 enabled_when 属性的用法

7.2.4　配置视图

前面介绍了如何使用 Item 和 Group 等对象组织窗口界面中的内容，本节介绍如何配置窗口本身的属性。

通过 kind 属性可以修改 View 对象的显示类型：

- 'modal'：模态窗口, 非即时更新。
- 'live'：非模态窗口，即时更新。
- 'livemodal'：模态窗口，即时更新。
- 'nonmodal'：非模态窗口，非即时更新。
- 'wizard'：向导类型。
- 'panel'、'subpanel'：嵌入到其他窗口中的面板，即时更新，非模态窗口。

其中，'modal'、'live'、'livemodal'、'nonmodal'这 4 种类型的 View 对象都采用窗口界面来显示内容。所谓模态窗口，是指在窗口关闭之前，程序中的其他窗口都不能被激活。而即时更新则是指，当窗口中编辑器的内容改变时，会立即反映到编辑器所对应的模型对象的属性值。非即时更新的窗口则会复制模型对象，所有的改变在副本上进行，只有当用户单击 OK 或 Apply 按钮确定修改时，才会修改原始模型对象的属性。

'wizard'由一系列特定的向导窗口组成，属于模态窗口，并且即时更新数据。

'panel'和'subpanel' 则是嵌入到窗口中的面板，'panel'可以拥有自己的命令按钮，而'subpanel'则没有命令按钮。

在对话框中经常可以看到 OK、Cacel、Apply 之类的按钮，我们称之为命令按钮，它们完成所有对话框都需要的一些共同操作。在 TraitsUI 中，这些按钮可以通过 View 对象的 buttons 属性进行设置，其值为要显示的按钮列表。

TraitsUI 中定义了 UndoButton、ApplyButton、RevertButton、OKButton、CancelButton 和 HelpButton 共 6 个标准的命令按钮，每个按钮对应一个名字。在指定 buttons 属性时，可以使用按钮的类名或者它对应的名字。与按钮类对应的名字就是类名除去 Button，例如 UndoButton 对应为"Undo"。

enthought.traits.ui.menu 中还预定义了一些命令按钮，以方便用户直接使用整套按钮：

```
OKCancelButtons = [OKButton, CancelButton ]
ModalButtons = [ ApplyButton, RevertButton, OKButton, CancelButton, HelpButton ]
LiveButtons = [ UndoButton, RevertButton, OKButton, CancelButton, HelpButton ]
```

7.3 用 Handler 控制界面和模型

虽然 TraitsUI 库的界面设计使用 MVC 模式，但是在前面的介绍中，没有出现过任何有关控制器(Controller)的代码。这是因为 TraitsUI 库提供了默认的控制器。我们可以通过继承 Handler 类，创建自己的控制器类，从而对视图和模型进行更自由的控制。

控制器的最主要功能是界面的事件处理，这些事件要么对模型对象进行修改，要么对界面进行修改。模型、视图以及控制器都是单独的实体。一个视图对象可以为多个模型对象生成界面，同样，一个控制器也可以用来处理不同视图界面中所产生的事件。因此控制器和视图以及模型之间不存在静态的联系。但是为了让控制器能够真正起作用，它必须知道自己所要处理的

视图和模型。在 TraitsUI 中，控制器使用 UIInfo 对象获得它所对应的视图和模型。

当 TraitsUI 从视图创建界面时，将创建一个 UIInfo 对象，其中包括界面和模型对象的引用。当界面事件引起控制器的方法被调用时，这个 UIInfo 对象会作为参数传递给被调用的方法。

TraitsUI 提供了三种指定控制器的方法：

- 将控制器作为视图的属性：使用视图对象的 handler 属性指定控制器对象，此视图所产生的界面都使用它进行事件处理。
- 显示界面时设置：调用 edit_traits()、configure_traits()或 ui()等方法显示界面时，将控制器对象传递给 handler 参数。它比视图的 handler 属性有更高的优先级，将覆盖 handler 属性的设置。
- 将视图作为控制器的一部分进行定义。

7.3.1 用 Handler 处理事件

当显示某个视图时，将按照下面的顺序执行控制器中的方法：

(1) 创建一个 UIInfo 对象。

(2) 执行控制器的 init_info()方法。

(3) 创建一个 UI 对象以表示实际的窗口。

(4) 执行控制器的 init()方法。

(5) 执行控制器的 position()方法。

(6) 显示实际的窗口。

除了上面的 init_info()、init()、position()之外，当用户对界面进行操作时，还会执行如下的方法：

- apply()：用户单击窗口中的 Apply 按钮，在模型对象的数据更新之后。
- close()：用户关闭窗口，在窗口关闭之前。
- closed()：在窗口关闭之后。
- revert()：用户单击了 Revert 或 Cancel 按钮。
- setattr()：用户通过界面修改了模型对象的某个 Trait 属性。
- show_help()：用户单击了窗口中的 Help 按钮。

下面通过一个实例演示上述各个方法的用法：



traitsUI_handler.py
用 Handler 的方法处理各种事件

```
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group, Handler
from enthought.traits.ui.menu import ModalButtons

g1 = [Item('department', label=u"部门"),
      Item('name', label=u"姓名")]
```

```
g2 = [Item('salary', label=u"工资"),
      Item('bonus', label=u"奖金")]

class Employee(HasTraits):
    name = Str
    department = Str
    salary = Int
    bonus = Int

    def _department_changed(self): ❶
        print self, "department changed to ", self.department

    def __str__(self): ❷
        return "<Employee at 0x%x>" % id(self)

view1 = View(
    Group(*g1, label = u'个人信息', show_border = True),
    Group(*g2, label = u'收入', show_border = True),
    title = u"外部视图",
    kind = "modal",   ❸
    buttons = ModalButtons
)

class EmployeeHandler(Handler): ❹
    def init(self, info):
        super(EmployeeHandler, self).init(info)
        print "init called"

    def init_info(self, info):
        super( EmployeeHandler, self).init_info(info)
        print "init info called"

    def position(self, info):
        super(EmployeeHandler, self).position(info)
        print "position called"

    def setattr(self, info, obj, name, value):
        super(EmployeeHandler, self).setattr(info, obj, name, value)
        print "setattr called:%s.%s=%s" % (obj, name, value)

    def apply(self, info):
        super(EmployeeHandler, self).apply(info)
        print "apply called"

    def close(self, info, is_ok):
        super(EmployeeHandler, self).close(info, is_ok)
        print "close called: %s" % is_ok
```

```
        return True

    def closed(self, info, is_ok):
        super(EmployeeHandler, self).closed(info, is_ok)
        print "closed called: %s" % is_ok

    def revert(self, info):
        super(EmployeeHandler, self).revert(info)
        print "revert called"

if __name__ == "__main__":
    zhang = Employee(name="Zhang")
    print "zhang is ", zhang
    zhang.configure_traits(view=view1, handler=EmployeeHandler()) ❺
```

❶在 Employee 模型类中，定义了 department 属性的事件处理方法_department_changed()。❷覆盖标准的字符串转换方法__str__()，用以显示模型对象占据的地址[3]。

❸为了显示对话框的标准按钮，在创建视图对象时设置 View 的 kind 和 button 参数分别为'modal'和 ModalButtons。这样一来，界面所显示的对话框便是“模态窗口、非即时更新”，并且有 Apply、Revert、OK、Cancel、Help 等按钮，如图 7-7 所示。

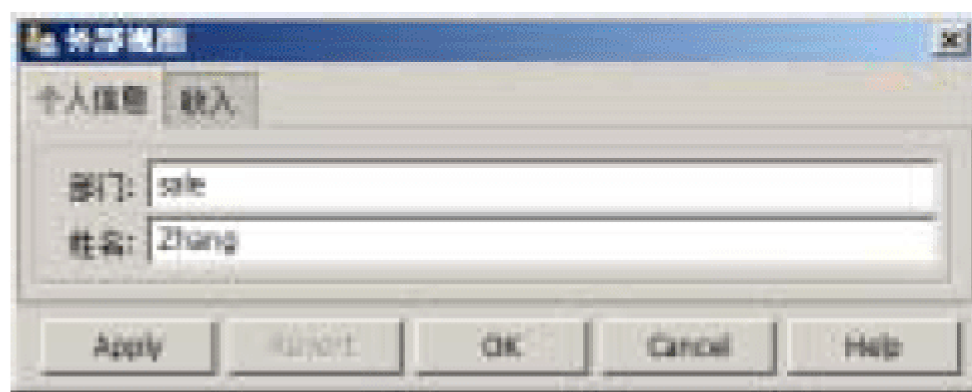


图 7-7　带标准按钮的模式对话框

❹EmployeeHandler 从 Handler 继承，它覆盖了 Handler 中的 init、init_info、position、setattr、apply、close、closed、revert 等方法。如果 close()返回 True，窗口会被关闭；如果返回 False，就不会关闭窗口。在这些方法中，将首先调用父类中被覆盖的方法，从而实现默认的控制器的功能。实际上，父类 Handler 中的大部分方法不执行任何任务，因此也可以不运行它们，请读者阅读 Handler 类的源代码以了解每个方法的默认功能。

❺最后，创建一个控制器对象，并将它传递给 configure_traits()的 handler 参数。

在 IPython 或命令行中运行此程序，在对话框的“部门”文本框中输入“sale”，然后单击 Apply 按钮，最后单击 OK 按钮关闭对话框。程序会在命令行窗口中输出控制器的各个方法的调用情况。

首先，对象 zhang 所表示的对象地址为“0x21794b0”：

```
zhang is  <Employee at 0x21794b0>
```

③ 使用 Python 的内置函数 id()可以获取对象的地址，地址相同的两个对象实际上是同一个对象。

调用 zhang 的 configure_traits()之后，在窗口显示之前，调用了 init_info()、init()、position()三个方法：

```
init info called
init called
position called
```

接下来输入“sale”，每次输入一个字符，都会修改模型对象的 department 属性，从而调用模型对象的_department_changed()，接着会调用控制器的 setattr()。因此控制器的 setattr()是在模型数据更新之后被调用的。

```
<Employee at 0x29d60f0> department changed to  s
setattr called:<Employee at 0x29d60f0>.department=s
<Employee at 0x29d60f0> department changed to  sa
setattr called:<Employee at 0x29d60f0>.department=sa
<Employee at 0x29d60f0> department changed to  sal
setattr called:<Employee at 0x29d60f0>.department=sal
<Employee at 0x29d60f0> department changed to  sale
setattr called:<Employee at 0x29d60f0>.department=sale
```

setattr()的调用参数如下：

```
setattr(self, info, obj, name, value)
```

其中，info 是 UIInfo 对象，obj 是被修改属性的模型对象，name 是被修改的属性名，而 value 是被修改之后的值。仔细比较前面输出的对象地址就会发现：被修改了属性的模型对象的地址并不是对象 zhang 的地址。这是因为“非即时更新”的对话框会对一个副本对象进行修改，在单击 Apply 或 OK 按钮时，才会将副本的内容写回原对象；而对副本对象的修改是“即时更新”的。

当单击 Apply 按钮时，程序输出了以下内容，我们看到：原始对象的 department 属性也被更新为“sale”了。

```
setattr called:<Employee at 0x29d60f0>.department=sale
<Employee at 0x21794b0> changed to  sale
apply called
```

而 setattr()再次被调用的原因是：单击 Apply 按钮时，"部门"文本框会失去焦点，从而引起 setattr()被调用。读者可以交替单击界面中的两个文本框，观察失去焦点时的 setattr()调用。由于此时模型对象的 department 属性没有变化，因此_department_changed()不会被调用。

最后单击 OK 按钮，程序输出下面两行，其中的 True 是 is_ok 参数的值，True 表示用户单击的是 OK 按钮，如果单击 Cancel 按钮，输出的将为 False：

```
close called: True
closed called: True
```

7.3.2 Controller 和 UIInfo 对象

Handler 类中每个事件处理方法的第一个参数都是 UIInfo 对象，通过它可以获得控制器对应的模型对象和视图对象所产生的界面。但是有时我们希望可以通过控制器的属性访问它们。TraitsUI 提供了从 Handler 继承的 Controller 类，它有两个 Trait 属性：model 和 info，分别保存模型对象和 UIInfo 对象。下面通过实例介绍 Controller 的用法以及 UIInfo 对象的内容。



traitsUI_UIInfo.py
使用 Controller 观察 UIInfo 对象

程序中的模型和视图与上一节的 “traitsUI_handler.py”相同。创建模型对象、控制器以及显示界面的代码如下：

```
zhang = Employee(name="Zhang")
c = Controller(zhang)
c.configure_traits(view=view1)
```

在创建 Controller 控制器时使用模型对象进行初始化，之后可以通过 c.model 访问此模型对象。调用 Controller 对象的 configure_traits()，不会显示控制器本身的 Trait 属性编辑窗口，而是显示模型对象的属性编辑窗口。

在开启“-wthread”选项的 IPython 命令行中，执行下面的语句之后，就可以交互式地查看控制器 c 的内容了：

```
>>> run traitsUI_UIInfo.py
>>> c
>>> <enthought.traits.ui.handler.Controller object at 0x032F58D0>
```

由于无论是控制器类、视图类还是模型类，最终都是从 HasTraits 类继承，因此可以调用 get()来快速查看其内容：

```
>>> c.get()
{'info': <enthought.traits.ui.ui_info.UIInfo object at 0x036097E0>,
 'model': <__main__.Employee object at 0x035BE9C0>}
>>> c.info.get()
{'initialized': True,
'ui': <enthought.traits.ui.ui.UI object at 0x035BEC30>}
>>> c.info.ui.get()
[[省略]]
```

我们看到，c.info 是一个 UIInfo 对象，而 UIInfo 对象中最重要的内容就是 UI 对象 c.info.ui。

UI 对象中保存有用户界面的各种信息。对 UI 对象的详细介绍已超出本书的范围，感兴趣的读者可自行查看其源代码。下面简要地查看 UI 对象的几个属性：

```
>>> ui = c.info.ui
>>> ui.title = "hello" # 修改窗口的标题
>>> ui.context # 存储和界面相关的模型、控制器对象的字典
{'controller': <enthought.traits.ui.handler.Controller object at 0x035BECF0>,
 'handler': <enthought.traits.ui.handler.Controller object at 0x035BECF0>,
 'object': <__main__.Employee object at 0x035BE9C0>} # 模型对象
>>> ui.control # ui 对象所表示的实际界面控件，它是 wx 库的一个 Frame 对象，即窗口
<wx._windows.Frame; proxy of <Swig Object of type 'wxFrame *' at 0x33b3668> >
>>> ui.view # 创建 ui 对象的视图对象
( Group(
    Item( 'department'
          label = u'\u90e8\u95e8'
[[省略]]
>>> ui._editors # 界面中编辑 Trait 属性用的编辑器列表
[<enthought.traits.ui.wx.text_editor.SimpleEditor object at 0x035F3A50>,
[[省略]]
```

7.3.3 响应 Trait 属性的事件

前面曾介绍过，在从 HasTraits 继承的模型类中，可以通过定义_traitname_changed()来响应 traitname 属性值改变的事件。这是在模型类中响应事件，如果要在控制器类中响应，可以通过定义 setattr()，响应用户通过界面对模型对象的 Trait 属性进行修改的事件。

如果希望仅响应模型对象中某个特定属性的事件，可以在控制器类中定义如下格式的事件响应方法：

```
extended_traitname_changed(self, info)
```



setattr()只有在通过界面修改模型对象的属性时才会被调用。但是，只要模型对象的 traitname 属性被修改，extended_traitname_changed()就会被调用，而不管是否是通过界面进行修改。

其中的 extended 是视图所产生的 UI 对象的 context 属性中与模型对象相对应的键，通常都为'object'。如果 UI 对象的 context 属性是由 context 参数指定，那么 extended 可以是其中的任何一个模型对象所对应的键，在实例“traits_multi_models.py”中，它可以是'p1'或'p2'。

这样的事件响应方法在界面窗口初始化时，以及在对应的属性改变时都会被调用。为了区分二者，可以对 info 参数的 initialized 属性进行判断。下面是一个例子：



traitsUI_handler_event.py
在控制器中响应模型对象特定属性的事件

```
from enthought.traits.api import HasTraits, Bool
from enthought.traits.ui.api import View, Handler

class MyHandler(Handler):
    def setattr(self, info, object, name, value): ❶
        Handler.setattr(self, info, object, name, value)
        info.object.updated = True ❷
        print "setattr", name

    def object_updated_changed(self, info): ❸
        print "updated changed", "initialized=%s" % info.initialized
        if info.initialized:
            info.ui.title += "*"

class TestClass(HasTraits):
    b1 = Bool
    b2 = Bool
    b3 = Bool
    updated = Bool(False)

view1 = View('b1', 'b2', 'b3',
            handler=MyHandler(),
            title = "Test",
            buttons = ['OK', 'Cancel'])

tc = TestClass()
tc.configure_traits(view=view1)
```

❶MyHandler 类中定义了 setattr()方法，在通过界面修改了模型对象的任何一个 Trait 属性之后，它将被调用。❷在 setattr()中修改模型对象的 updated 属性为 True。

❸当模型对象的 updated 属性被修改时，它在控制器对象中对应的 object_updated_changed()将被调用。由于 updated 属性不是通过界面修改的，因此这时 setattr()不会被再次调用。

在 IPython 下执行此程序之后，直接修改模型对象 tc 的 b1 属性：

```
>>> run traitsUI_handler_event.py
updated changed initialized=False # 初始化界面时，将调用一次
>>> tc.b1 = True # 直接修改模型对象的属性，不会调用控制器的 setattr 方法
```

在界面中选中“B2”复选框，IPython 中将输出如下内容：

```
updated changed initialized=True
setattr b2
```

程序的执行过程如图 7-8 所示。左图为界面显示之后的初始状态。如中图所示，直接修改模型对象的属性不会触发控制器的 setattr()运行，但是“B1”复选框会改为被选中；当通过界面选中“B2”复选框之后，将调用控制器的 setattr()，从而设置模型对象的 update 属性为 True，这会再次触发控制器的 object_updated_changed()运行，从而在窗口标题中添加“*”。

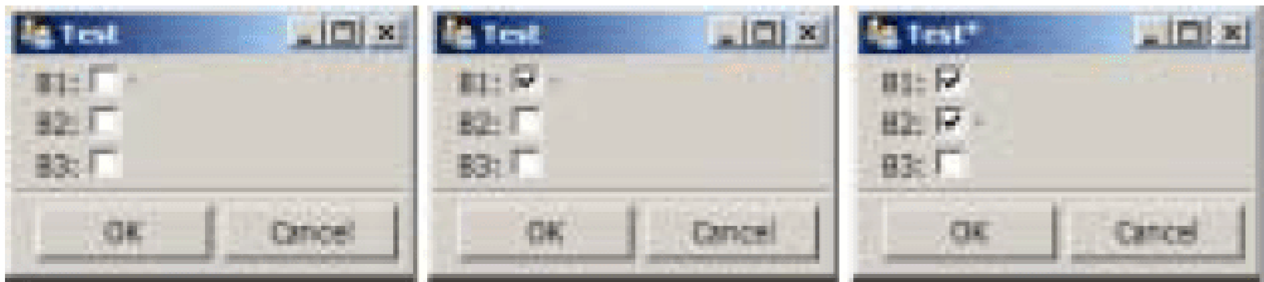


图 7-8　界面初始状态(左)，直接修改 tc.b1 之后(中)，选中“B2”复选框之后(右)

7.4　属性编辑器

每个 Trait 类型都有一种默认的界面编辑器(控件)与之对应，如果在视图对象中不指定编辑器，将使用默认的编辑器生成界面。每种编辑器都可以有 4 种样式："simple"、"custom"、"text"和"readonly"：

- "simple"：默认值，使用一个比较简单的编辑器，尽量少占用界面空间。
- "custom"：使用较复杂的编辑器，尽量呈现更多的内容。
- "text"：使用一个文本编辑器。
- "readonly"：使用只读控件显示。

TraitsUI 提供了丰富的编辑器库，以至于我们很少有自己设计编辑器的需求。由于 TraitsUI 的编辑器种类繁多，本书不能一一对其进行详细介绍，感兴趣的读者可运行 Python(x,y)的文档目录下的演示程序，运行界面如图 7-9 所示。

c:\pythonxy\doc\Enthought Tool Suite\Traits\examples\demo\demo.py
TraitsUI 演示程序

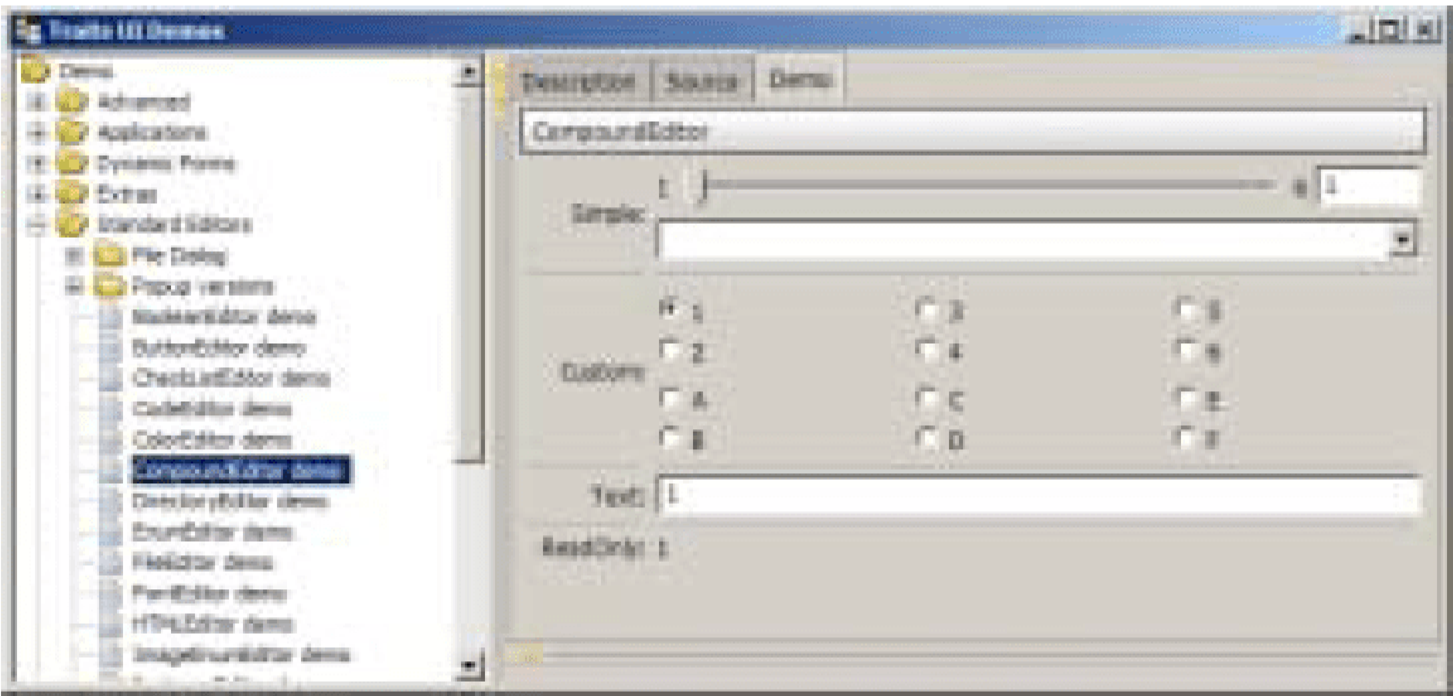


图 7-9　TraitsUI 演示程序的运行界面

下面将以几个实例简单地介绍如何使用 TraitsUI 提供的编辑器。

7.4.1 编辑器演示程序

由于 TraitsUI 提供了众多的编辑器，并且每个编辑器又有许多属性配置，对它们一一进行介绍将是一件枯燥无味的事情。因此本节介绍一个能显示各种编辑器效果的演示程序，图 7-10 是它的界面截图。界面的左半部分是用来创建各种 Trait 属性的源程序列表，对于选中的某个 Trait 属性，在界面的右半部分将使用"simple"、"custom"、"text"和"readonly"4 种样式创建属性编辑器。

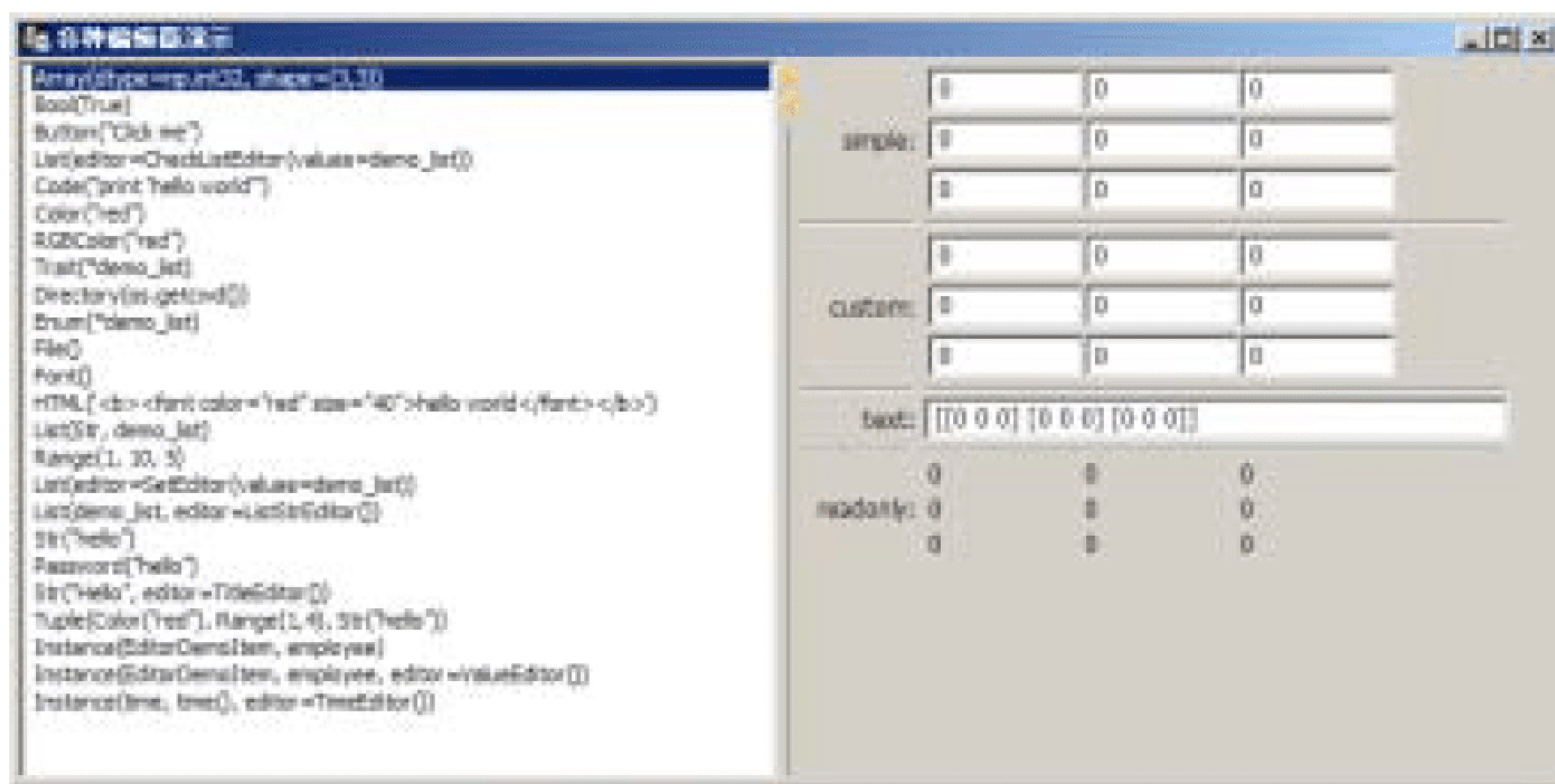


图 7-10 各种编辑器的效果演示



traitsUI_editors.py
各种编辑器的效果演示

```
class EditorDemo(HasTraits):
    codes = List(Str)
    selected_item = Instance(EditorDemoItem)
    selected_code = Str ❶
    view = View(
        HSplit(
            Item("codes", style="custom", show_label=False,
                editor=ListStrEditor(editable=False, selected="selected_code")), ❶
            Item("selected_item", style="custom", show_label=False),
        ),
        resizable=True,
        width = 800,
        height = 400,
        title=u"各种编辑器演示"
```

```
        )

    def _selected_code_changed(self):
        item = EditorDemoItem(code=self.selected_code)
        item.add_trait("item", eval(self.selected_code)) ❷
        self.selected_item = item
```

上面是创建主界面的程序。在 EditorDemo 类中，codes 属性保存一组用来创建各种 Trait 属性的字符串，selected_item 属性是 EditorDemoItem 的对象。在 EditorDemo 类的视图定义中，使用 HSplit 将 codes 和 selected_items 所对应的编辑器进行水平分割。❶用 editor 参数设置 codes 属性的编辑器为 ListStrEditor，它是一个显示一组字符串的列表框控件。其 editable 属性为 False，表示列表框中的字符串都是只读的。selected 属性是保存被选中的字符串的 Trait 属性名，在 EditorDemo 类中用 selected_code 属性保存列表框中被选中的字符串。可以通过 editor 参数设置 Item 对象的编辑器，这样界面中将使用指定的编辑器显示 Trait 属性。

❷当用户通过列表框选中了某个字符串时，selected_code 属性将发生变化，因此_selected_code_changed()将被调用。这里创建一个 EditorDemoItem 对象，并调用其 add_trait()方法，动态地为其创建一个名为 item 的 Trait 属性，其类型则通过 eval()对 selected_code 字符串进行求值获得。这段代码通过字符串动态地创建 Trait 类型，并使用此类型动态地创建 Trait 属性。

```
class EditorDemoItem(HasTraits):
    code = Code()
    view = View(
        Group(
            Item("item", style="simple", label="simple", width=-300), ❶
            "_",  ❷
            Item("item", style="custom", label="custom"),
            "_",
            Item("item", style="text", label="text"),
            "_",
            Item("item", style="readonly", label="readonly"),
        ),
    )
```

在 EditorDemoItem 类的视图定义中，我们使用 4 种样式为其 item 属性定义编辑器。请注意在 EditorDemoItem 类的定义中并没有 item 属性，但是由于视图中使用属性名字符串定义编辑器，因此只有在真正使用视图创建界面时，才会去访问 item 属性，这时已经通过 add_trait()为其添加了 item 属性。❶Item 对象的 width 属性可以指定编辑器的宽度，以像素点为单位的长度用整数表示，负数表示强制设置其宽度。width 属性还有多种设置宽度的用法，请读者查看 Item 类源代码中的注释。❷使用下划线字符串在界面中创建分割线。

```
employee = Employee()
```

```
demo_list = [u"低通", u"高通", u"带通", u"带阻"]

trait_defines ="""
    Array(dtype="int32", shape=(3,3))
    Bool(True)
    Button("Click me")
    List(editor=CheckListEditor(values=demo_list))
    Code("print 'hello world'")
    Color("red")
    RGBColor("red")
    Trait(*demo_list)
    Directory(os.getcwd())
    Enum(*demo_list)
    File()
    Font()
    HTML('<b><font color="red" size="40">hello world</font></b>')
    List(Str, demo_list)
    Range(1, 10, 5)
    List(editor=SetEditor(values=demo_list))
    List(demo_list, editor=ListStrEditor())
    Str("hello")
    Password("hello")
    Str("Hello", editor=TitleEditor())
    Tuple(Color("red"), Range(1,4), Str("hello"))
    Instance(EditorDemoItem, employee)
    Instance(EditorDemoItem, employee, editor=ValueEditor())
    Instance(time, time(), editor=TimeEditor())
"""
demo = EditorDemo()
demo.codes = [s.split("#")[0].strip() for s in trait_defines.split("\n") if s.strip()!=""]
demo.configure_traits()
```

最后是定义各种 Trait 类型的程序。我们可以在定义 Trait 类型时，通过 editor 参数设置其对应的编辑器，这样就不需要在视图的 Item 对象中定义了。

请读者自行研究每个 Trait 类型的定义以及它们所创建的界面控件，这里就不再进行详细说明了。

7.4.2 对象编辑器

随着程序开发的进行，界面中的控件数目会逐渐增多，功能会越来越复杂，这意味着与界面对应的模型类也会变得复杂起来。为了便于代码的理解、管理以及重用，我们需要对模型类及其对应的界面视图对象进行重构。将程序中重复使用、相对独立的部分作为组件分离出来，单独为其设计模型类和视图对象，最终的应用程序将由一系列这样的组件构成。这些组件可以

在程序的不同地方重复使用，从而起到功能分离、代码重用等多方面的作用。TraitsUI 的 MVC 模式非常适合这种组件开发方式，下面让我们通过一些实例深入理解 MVC 模式所带来的便利。

> traitsUI_component.py
> 组件演示程序

此实例程序将创建一个如图 7-11 所示的界面，用户可以通过上方的下拉列表框选择一种形状，下方的控件会自动根据所选的形状发生变化，当通过这些控件输入形状数据时，界面下方的信息栏会自动进行更新。由于程序较长，下面我们将它分为几个部分进行分析。

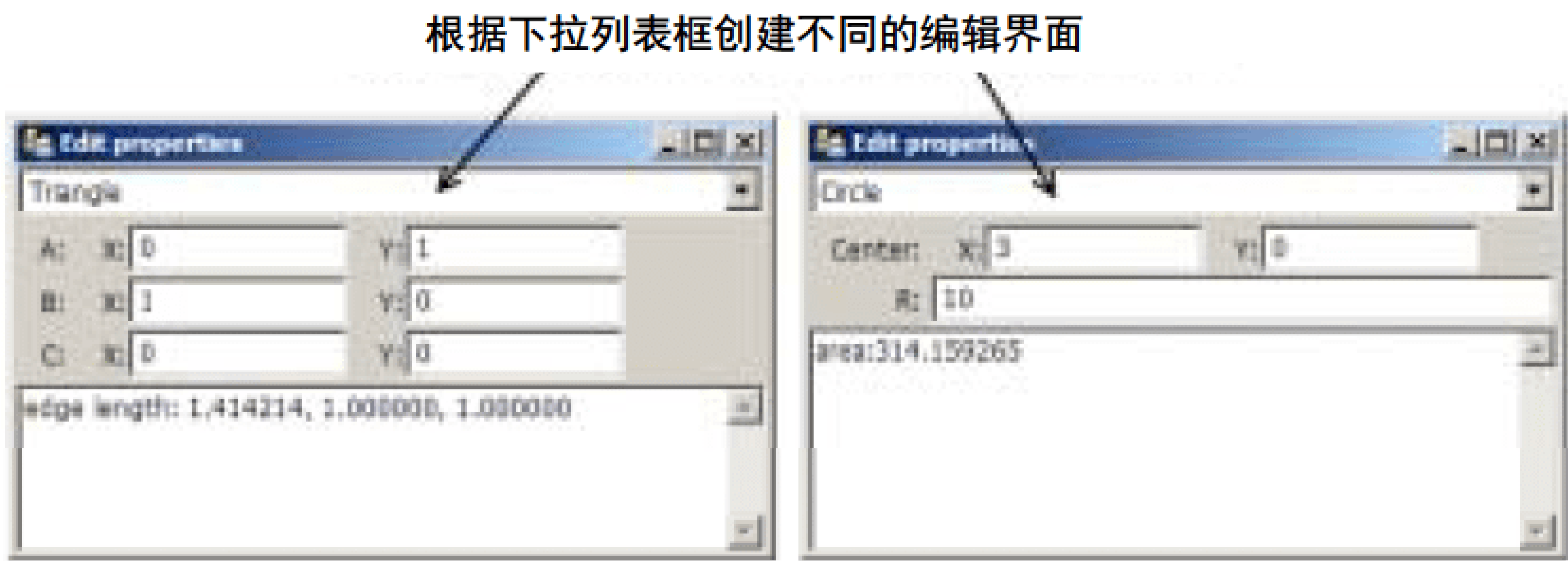


图 7-11　组件演示，根据下拉列表框创建不同的编辑界面

```
class Point(HasTraits):
    x = Int
    y = Int
    view = View(HGroup(Item("x"), Item("y")))
```

上面的程序定义了用于保存平面上点的坐标的 Point 类。我们还为它指定了一个视图对象，视图中 X 和 Y 轴的坐标值输入框是横向排列的。在 IPython 中运行下面的语句即可看到 Point 对象所创建的界面效果：

```
>>> Point().configure_traits()
```

我们可以将 Point 类当做组件使用，将它嵌入到更复杂的界面中去。在下面的程序中定义了一个基类 Shape，以及两个派生类 Triangle 和 Circle，并使用 Point 类定义所有表示二维坐标点的属性：

```
class Shape(HasTraits):
    info = Str ❶

    def __init__(self, **traits):
        super(Shape, self).__init__(**traits)
        self.set_info() ❷
```

```
class Triangle(Shape):
    a = Instance(Point, ()) ❸
    b = Instance(Point, ())
    c = Instance(Point, ())

    view = View(
        VGroup(
            Item("a", style="custom"), ❹
            Item("b", style="custom"),
            Item("c", style="custom"),
        )
    )

    @on_trait_change("a.[x,y],b.[x,y],c.[x,y]")
    def set_info(self):
        a,b,c = self.a, self.b, self.c
        l1 = ((a.x-b.x)**2+(a.y-b.y)**2)**0.5
        l2 = ((c.x-b.x)**2+(c.y-b.y)**2)**0.5
        l3 = ((a.x-c.x)**2+(a.y-c.y)**2)**0.5
        self.info = "edge length: %f, %f, %f" % (l1,l2,l3)

class Circle(Shape):
    center = Instance(Point, ())
    r = Int

    view = View(
        VGroup(
            Item("center", style="custom"),
            Item("r"),
        )
    )

    @on_trait_change("r")
    def set_info(self):
        self.info = "area:%f" % (pi*self.r**2)
```

❶在 Shape 类中定义了一个 info 属性，❷在初始化方法中调用派生类的 set_info()以修改 info 属性。

❸在 Triangle 类中使用“Instance(Point, ())”定义了表示三角形三个顶点坐标的属性：a、b 和 c。在 Circle 类中使用同样的方式定义了表示圆心坐标的 center 属性。这些属性的值都是 Point 对象。Instance 的第二个参数指定创建默认对象时所用的参数，当没有第二个参数时，它所定义的属性的默认值为 None，这里用一个空元组表示与其对应的属性的默认值是通过调用“Point()”得到的，即默认为 Point()所创建的 Point 对象。

❹如果 Trait 属性是 Instance 类型，并且它在视图中对应的编辑器为"custom"样式，那么属

性对象的视图将直接嵌入到当前的视图中。因此，在 Triangle 和 Circle 对象的编辑界面中将嵌入多个 Point 对象的编辑器。在 IPython 中运行下面的程序可以看到所创建的界面效果：

```
>>> Triangle().configure_traits()
>>> Circle().configure_traits()
```

接下来，使用上面的形状类创建最终的形状选择类 ShapeSelector：

```
class ShapeSelector(HasTraits):
    select = Enum(*[cls.__name__ for cls in Shape.__subclasses__()]) ❶
    shape = Instance(Shape) ❷

    view = View(
        VGroup(
            Item("select"),
            Item("shape", style="custom"), ❸
            Item("object.shape.info", style="custom"), ❹
            show_labels = False
        ),
        width = 350, height = 300, resizable = True
    )

    def __init__(self, **traits):
        super(ShapeSelector, self).__init__(**traits)
        self._select_changed()

    def _select_changed(self):
        klass =  [c for c in Shape.__subclasses__() if c.__name__ == self.select][0]
        self.shape = klass() ❺
```

❶下拉列表框所对应的 select 属性的类型为 Enum(枚举)，为了让程序显得更自动化一些，我们不直接指定枚举类型的候选值，而是通过 Shape 的派生类名创建候选值列表。这样一来，当我们添加其他的 Shape 派生类时，便不需要对这段代码进行任何修改。

❷shape 属性的类型是 Shape，由于这里不需要创建默认的 Shape 对象，因此不用指定 Instance 的第二个参数。❺当 select 属性发生变化时，在其事件处理方法_select_changed()中，创建 select 属性所对应的类的对象并赋值给 shape 属性。❸shape 属性所对应的编辑器是"custom"样式，因此将它的编辑界面作为组件嵌入到 ShapeSelector 的界面中。并且它能根据当前的 shape 属性值，动态更新界面上的编辑器。也就是说，当 shape 属性是 Triangle 对象时，将使用 Triangle 类的视图创建编辑器；而当 shape 属性是 Cirlce 对象时，将使用 Circle 类的视图创建编辑器。

❹通过"object.shape.info"可以为 shape 属性的 info 属性在界面中创建编辑器。当为某个 Trait 属性的属性创建编辑器时，需要在属性名的前面添加"object."。

读者也许会认为这种为属性的属性创建编辑器的做法有些混乱，一个比较简单的解决方法

就是从 ShapeSelector 的视图中删除"object.shape.info"的 Item 对象，并分别给 Triangle 和 Circle 的视图添加显示 info 属性的编辑器：Item("info", style="custom")。这种做法的缺点是，需要给每个从 Shape 派生的类的视图添加 info 属性的编辑器，而当我们不想显示 info 属性时，代码的修改量也会随着 Shape 的派生类的增加而增加。

还有一种使用多个视图对象的方法，它充分体现了 MVC 模式将模型和视图完全分离的优点。



traitsUI_component_multi_view.py
使用多个视图显示组件

由于程序的改动不大，我们只介绍它和“traitsUI_component.py”的不同之处。

```
class Shape(HasTraits):
    info = Str
    view_info = View(Item("info", style="custom", show_label=False))
```

首先，为 Shape 类添加一个 view_info 视图，专门用于显示其 info 属性。这样一来，Shape 的派生类 Triangle 和 Circle 便都具有两个视图：view、view_info。如果模型类有多个视图，那么当将其嵌入到其他视图中时，需要指定使用哪个视图创建编辑器。因此，ShapeSelector 类的视图需要进行如下修改：

```
view = View(
    VGroup(
        Item("select", show_label=False),
        VSplit( ❶
            Item("shape", style="custom", editor=InstanceEditor(view="view")), ❷
            Item("shape", style="custom", editor=InstanceEditor(view="view_info")),
            show_labels = False
        )
    ),
    width = 350, height = 300, resizable = True
)
```

❶为了和前面的例子有所区别，这里用一个垂直分隔容器将形状数据输入界面和显示形状信息的控件分隔开。❷shape 属性的编辑器样式仍然为"custom"，但是为了指定编辑器所使用的视图，我们需要通过 editor 参数传递一个 InstanceEditor 对象。而通过 InstanceEditor 对象的 view 参数可以指定创建界面时所使用的视图名。实际上，Instance 类型的 Trait 属性默认就是使用 InstanceEditor 作为"custom"样式的编辑器，因此前面的程序中都没有通过 editor 参数指定。当需要修改 InstanceEditor 对象的一些默认值时，就需要我们手工创建它了。

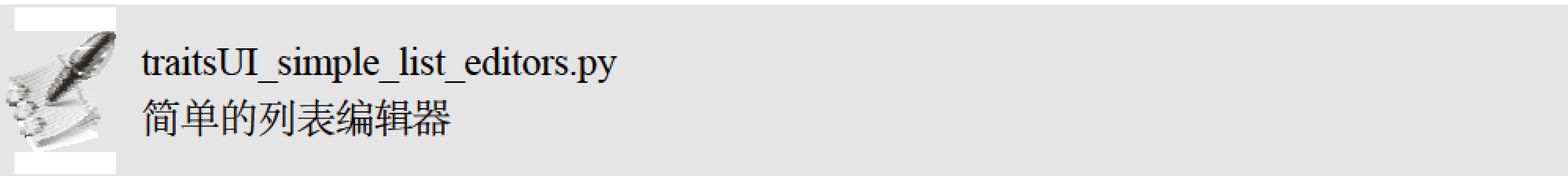
下面总结一下前面的内容：

- 通过将 Instance 类型的 Trait 属性的编辑器样式指定为"custom"，可以实现界面的层层嵌套，即组件功能。
- 当模型类有多个视图对象时，通过 InstanceEditor 的 view 参数可以选择其中的某个视图来创建编辑此模型对象的控件。

TraitsUI 的组件并不局限于界面上的某一块区域，我们可以在界面中的不同位置用不同的视图，为同一个模型对象创建多个不同的编辑器，因此使用 TraitsUI 创建的界面是非常灵活的。

7.4.3 字符串列表编辑器

在 Traits 库中，List 是表示列表的 Trait 类型。根据列表元素的类型，可以使用不同的编辑器显示其内容。让我们首先从最简单的字符串列表开始。下面的程序使用了三种不同的编辑器来显示字符串列表，界面如图 7-12 所示。



traitsUI_simple_list_editors.py
简单的列表编辑器

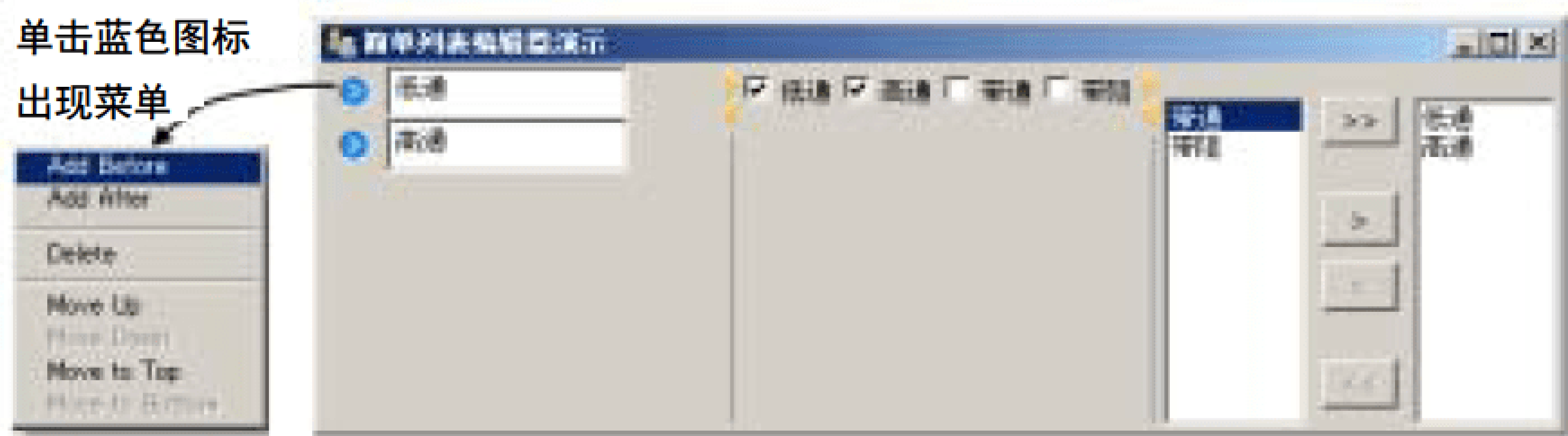


图 7-12　使用三种编辑器显示简单列表属性

```
filter_types = [u"低通", u"高通", u"带通", u"带阻"] ❶
class ListDemo(HasTraits):

    items = List(Str) ❷
    view = View(
        HSplit(
            Item("items", style="custom", show_label=False), ❸
            Item("items", style="custom",
                 editor=CheckListEditor(values=filter_types, cols=4)), ❹
            Item("items", editor=SetEditor(values=filter_types)), ❺
            show_labels=False
        ),
        resizable=True,
        width = 600,
        title=u"简单列表编辑器演示"
    )
```

❶首先创建一个列表 filter_types，它表示列表属性中元素的可能取值。❷定义 items 属性为字符串列表类型，这里的 Str 表示列表的元素类型必须是字符串。

❸使用默认的列表编辑器显示 items 属性，效果如图 7-12 左侧的编辑器所示。列表的每个元素使用一个文本编辑器进行编辑，在编辑器的左侧有一个蓝色的图标。单击此图标会弹出一个添加、删除和改变元素顺序的菜单。使用此编辑器可以为 items 属性添加任意字符串，不受候选值列表 filter_types 的限制。此外，指定 Item 对象的 style 参数为"custom"，让列表编辑器能够完全显示其内容。

❹通过 editor 参数指定编辑器为 CheckListEditor 对象。在创建 CheckListEditor 编辑器对象时，通过 values 和 cols 参数分别指定候选值列表和显示的列数。

CheckListEditor 编辑器会根据对应的 Item 对象的 style 属性来决定所使用的控件。这里选用"custom"样式，将使用复选框控件作为编辑器。如果样式为默认的"simple"，将会用下拉列表框作为编辑器。下拉列表框只能进行单选，因此它所对应的列表属性只能有一个元素，请读者修改演示程序以自行验证。显示效果如图 7-12 中间的编辑器所示。

❺使用 SetEditor 作为编辑器，由于 SetEditor 的"simple"和"custom"样式效果相同，因此不需要设置 style 属性。效果如图 7-12 右侧的编辑器所示。

使用其中任意一个编辑器修改属性值，其他的编辑器也会同时更新。由于 CheckListEditor 和 SetEditor 编辑器将 items 属性的内容锁定为有限的候选项，因此无法通过最左边的默认编辑器添加其他的字符串。

在上面的例子中，items 属性的候选值是固定的，在定义 ListDemo 类时，这些候选值就已经被确定，以后也无法改变。但是有时我们希望候选列表本身也可以动态地改变。为了实现这个目的，可以使用另外一个 Trait 属性保存候选列表：

```
class ListDemo2(HasTraits):
    filter_types = List(Str, value=[u"低通", u"高通", u"带通", u"带阻"]) ❶
    items = List(Str)
    view = View(
        HGroup(
            Item("filter_types", label=u"候选"), ❷
            Item("items", style="custom",
                editor=CheckListEditor(name="filter_types")), ❸
            show_labels=False
        ),
        resizable=True,
        width = 300,
        height = 180,
        title=u"动态修改候选值"
    )
```

❶定义一个 filter_types 属性来保存候选值列表，通过 value 参数指定初始值。❷在视图中使用默认的列表编辑器显示候选值属性，用户通过它可以动态地修改候选值列表的内容。❸在

创建 CheckListEditor 编辑器对象时，使用 name 参数指定候选值列表的属性名。图 7-13 是程序的运行效果。在此界面中，我们通过左边的列表编辑器添加了一个候选值：“均衡器”，右边的 CheckListEditor 编辑器的内容也同时发生了变化。

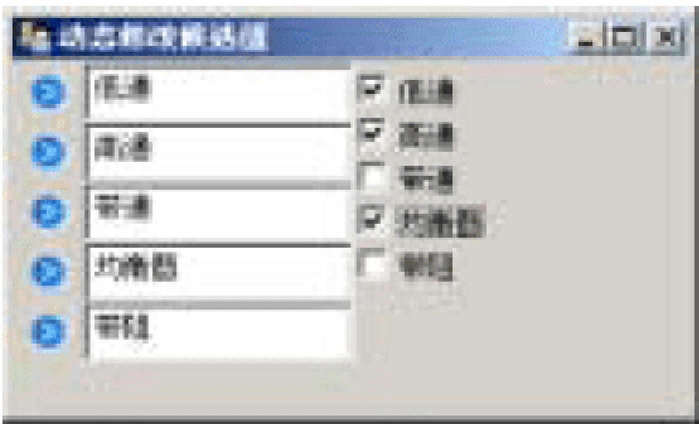


图 7-13　动态更改候选值列表

7.4.4　对象列表编辑器

当列表属性的元素是 HasTraits 对象时，还可以使用表格或标签页作为它的编辑器，效果如图 7-14 所示。

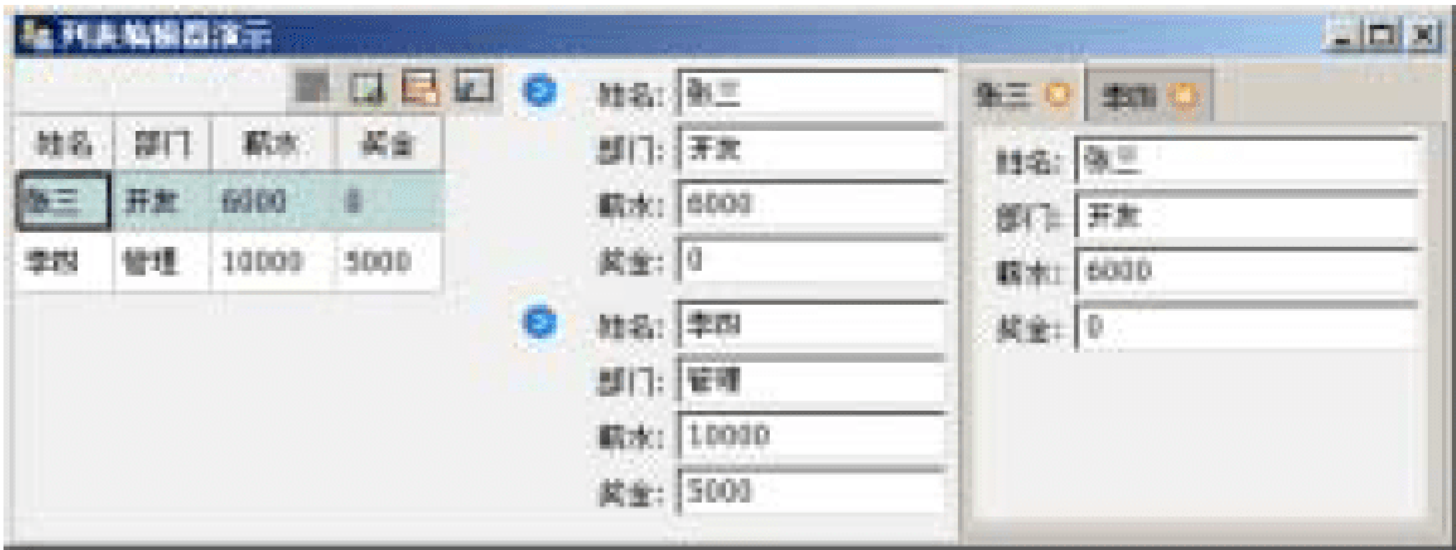


图 7-14　使用三种编辑器显示对象列表属性：表格(左)、列表(中)、标签页(右)

TraitsUI_list_editors.py
使用三种编辑器显示对象列表属性

```
from enthought.traits.api import HasTraits, Unicode, Int, List, Instance
from enthought.traits.ui.api import View, Item, TableEditor, ListEditor, HGroup
from enthought.traits.ui.table_column import ObjectColumn

class Employee(HasTraits):
    name = Unicode(label=u"姓名") ❶
    department = Unicode(label=u"部门")
    salary = Int(label=u"薪水")
    bonus = Int(label=u"奖金")
    view = View("name", "department", "salary", "bonus") ❷

table_editor = TableEditor(  ❸
```

```
        columns = [
            ObjectColumn(name="name", label=u"姓名"),
            ObjectColumn(name="department", label=u"部门"),
            ObjectColumn(name="salary", label=u"薪水"),
            ObjectColumn(name="bonus", label=u"奖金")
        ],
        row_factory = Employee,
        deletable = True,
        show_toolbar = True)

    list_editor = ListEditor(style="custom") ❻
    tab_editor = ListEditor(use_notebook=True, deletable=True, page_name=".name") ❼

    class EmployeeList(HasTraits):
        employees = List(Instance(Employee, factory=Employee)) ❸
        view = View(
            HGroup(
                Item("employees", editor=table_editor), ❹
                Item("employees", editor=list_editor),
                Item("employees", style="custom", editor=tab_editor),
                show_labels=False
            ),
            resizable=True,
            width = 600,
            title=u"列表编辑器演示"
        )
```

首先，Employee 是列表中对象的类型。在前面的例子中，界面中的标签在 Item 对象中定义。❶这里演示了另外一种定义标签的方法，可以在定义各个 Trait 属性时，使用 label 参数指定它们在视图中的标签。❷由于标签已经在 Trait 属性中定义，因此视图中可以不使用 Item，而是直接使用各个属性名的字符串。

在 EmployeeList 类中，employees 属性是一个 Employee 对象的列表。❸我们用 Instance (Employee)指定列表的元素类型为 Employee 对象。通过 Instance 的 factory 参数可以指定创建列表的默认对象的工厂函数，这里直接使用 Employee 类，也可以通过传递一个空元组来表示使用对应的类创建默认对象。此工厂函数将在图 7-14(中)的列表编辑器中用来给 employees 属性添加新的 Employee 对象。❹在视图中，我们使用三种编辑器显示 employees 属性。

❺用 TableEditor 创建表格编辑器，它的 columns 属性是一个 ObjectColumn 对象的列表，每个 ObjectColumn 对象描述模型对象中与其对应的 Trait 属性。row_factory 属性用于在表格中创建新的对象，这里也直接使用 Employee 类。deletable 属性为 True，表示可以删除表格中的行。show_toolbar 属性为 True，表示显示表格的工具栏，通过工具栏可以删除或添加行。

❻当 ListEditor 编辑器的 style 属性为"custom"时，列表中的每个模型对象都将使用与其对

应的视图来创建编辑器。❼当 ListEditor 编辑器的 use_notebook 属性为 True 时，将使用标签页显示列表中的多个模型对象。deletable 属性为 True，表示可以通过关闭标签页来从列表中删除模型对象。page_name 属性指定标签页标题所对应的 Trait 属性名，注意属性名的前面有一个“.”。



标签页的标题如果输入中文可能会出现错误，请按照下面的说明修改 TraitsUI 库的源代码。

本书写作时采用的 TraitsUI 库的版本为 3.6，如果在标签页标题中输入中文，会出现错误，这是因为 TraitsUI 中还有些代码对 Unicode 的支持不够，希望日后会有所改善。目前可以通过分析错误提示信息，修改 TraitsUI 库中相应的源程序。在下面的文件中搜索“'???'”：

```
site-packages\traitsbackendwx-3.6.0-py2.6.egg\enthought\traits\ui\wx\list_editor.py
```

将相应行的“str”改为“unicode”，有两处需要修改：

```
# 第一处
if name is None:
    name = str( xgetattr( view_object, # str 改为 unicode
                        self.factory.page_name[1:], '???' ) )
# 第二处
name = str( name ) or '???' # str 改为 unicode
```


7.5 菜单、工具条和状态栏

菜单、工具条和状态栏是窗口应用程序的标准组件，它们可以通过 View 对象的 menubar、toolbar 和 statusbar 属性指定。下面的程序演示了这一过程，效果如图 7-15 所示。



traitsUI_menu_toolbar.py
为界面添加菜单、工具条和状态栏

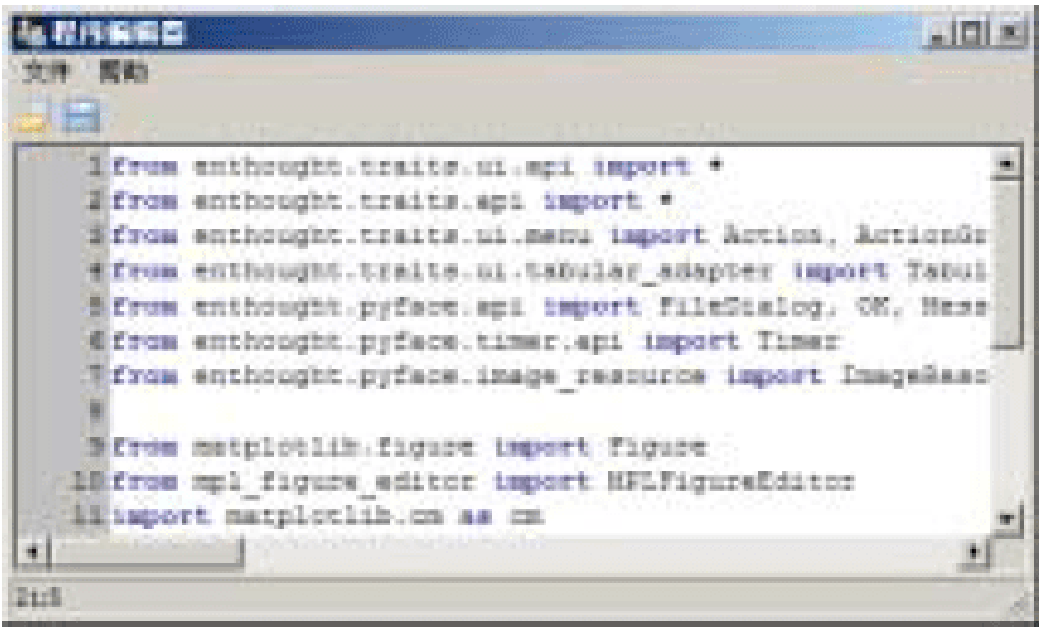


图 7-15　为界面添加菜单、工具条和状态栏

```
from enthought.traits.api import HasTraits, Code, Str, Int, on_trait_change
from enthought.traits.ui.api import View, Item, Handler, CodeEditor
from enthought.traits.ui.menu import Action, ActionGroup, Menu, MenuBar, ToolBar
from enthought.pyface.image_resource import ImageResource

class MenuDemoHandler(Handler):
    def exit_app(self, info):
        info.ui.control.Close()

class MenuDemo(HasTraits):
    status_info = Str
    current_line = Int
    text = Code
    def traits_view(self):
        file_menu = Menu( ❷
            ActionGroup(
                Action(id="open", name=u"打开", action="open_file"),
                Action(id="save", name=u"保存", action="save_file"),
            ),
            ActionGroup(
                Action(id="exit_app", name=u"退出", action="exit_app"),
            ),
            name = u"文件"
        )

        about_menu = Menu(
            Action(id="about", name=u"关于", action="about_dialog"),
            name = u"帮助"
        )

        tool_bar = ToolBar( ❸
            Action(
                image = ImageResource("folder_page.png", search_path = ["img"]),
                tooltip = u"打开文档",
                action = "open_file"
            ),
            Action(
                image = ImageResource("disk.png", search_path = ["img"]),
                tooltip = u"保存文档",
                action = "save_file"
            ),
        )

        return View(
            Item("text", style="custom", show_label=False,
```

```
                editor=CodeEditor(line="current_line")),
            menubar = MenuBar(file_menu, about_menu), ❶
            toolbar = tool_bar,
            statusbar = ["status_info"], ❹
            resizable = True,
            width = 500, height = 300,
            title = u"程序编辑器",
            handler = MenuDemoHandler()
        )

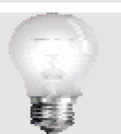
    @on_trait_change("text,current_line")
    def update_status(self):
        self.status_info = "%d:%d" % (self.text.count("\n")+1, self.current_line)

    def open_file(self):
        print "open_file"

    def save_file(self):
        print "save_file"

    def about_dialog(self):
        print "about_dialog"

demo = MenuDemo()
demo.configure_traits()
```



当视图的定义比较复杂时，可以定义名为 traits_view 的方法来返回视图对象。

❶菜单栏是一个 MenuBar 对象，它由多个 Menu 对象组成。❷每个 Menu 对象定义菜单栏中的一个菜单。在 Menu 对象中，使用 Action 对象定义菜单中的每个选项，并且可以使用 ActionGroup 对象对菜单进行分组。Action 对象的 name 属性是菜单上的文字，action 属性是对应的事件处理方法名，id 属性为其唯一标识。Action 类还有许多其他的属性，读者可以在 IPython 中输入如下命令来查看 Action 及其父类的源代码，以了解更多的属性用法：

```
>>> Action??
>>> Action.__base__??
```

❸工具条是一个 ToolBar 对象，它由多个 Action 对象组成，每个 Action 对象对应工具条中的一个按钮。通过 Action 对象的 image 属性可以指定工具按钮上显示的图标。为了方便指定图标文件的路径，我们使用 ImageResource 管理图标资源。

当用户使用菜单栏和工具条时，将运行 Action 对象的 action 属性所指定的事件处理方法。事件处理方法可以在模型类中定义，也可以在控制器类中定义。在上面的例子中，open_file()、save_file()、about_dialog()等在模型类中定义，而 exit_app()在控制器类中定义。因为在 exit_app()

中，需要调用后台界面库中窗口对象的 Close()方法来关闭应用程序。

❹状态栏则可以直接用一个字符串列表来指定，此列表中的每个字符串都是一个 Trait 属性名。在状态栏中将实时地显示对应的 Trait 属性值。

7.6 设计自己的编辑器

除了使用 TraitsUI 提供的标准编辑器之外，我们还可以使用自己编写的编辑器。这样可以根据应用程序的需求，制作出更加专业的界面。本节简要介绍 Trait 编辑器的工作原理，并且制作一个封装 matplotlib 绘图控件的编辑器，最后用它制作一个对 CSV 数据文件进行绘图的小工具。

Traits 库的路径

如果读者想深入了解 TraitsUI 的工作原理，需要查看它的源程序，因此首先需要知道它们在哪里。下面所有的路径都在 site-packages 目录下。

Traits：Traits-x.x.x-py2.6-win32.egg\enthought\traits，以下简称“%traits%”。

TraitsUI：Traits-x.x.x-py2.6-win32.egg\enthought\traits\UI，以下简称“%ui%”。

wx 后台界面库：TraitsBackendWX-x.x.x-py2.6.egg\enthought\traitsui\wx，以下简称“%wx%”。

7.6.1 Trait 编辑器的工作原理

先看下面的小程序，它定义了一个 TestStrEditor 类，其中有一个名为 test 的 Trait 属性，类型为 Str。在视图 view 中定义一个 Item 对象以显示 test 属性，但是没有通过 editor 参数指定它所使用的编辑器。当显示界面时，Traits 库将自动挑选文本框控件作为 test 属性的编辑器。



traitsUI_texteditor.py
字符串属性的默认编辑器

```
class TestStrEditor(HasTraits):
    test = Str
    view = View(Item("test"))
```

通过 Trait 类型对象的 create_editor()方法可以获得它的默认编辑器，例如：

```
>>> t = TestStrEditor()
>>> ed = t.trait("test").trait_type.create_editor()
>>> type(ed)
<class 'enthought.traits.ui.editors.text_editor.ToolkitEditorFactory'>
```

这个编辑器对象 ed 的类也是从 HasTraits 继承，因此可以调用 get()来显示它所有的 Trait 属性名和对应值：

```
>>> ed.get()
{'auto_set': True,
 'custom_editor_class': <class 'enthought.traits.ui.wx.text_editor.CustomEditor'>,
 'enabled': True,
 'enter_set': False,
 'evaluate': <enthought.traits.ui.editors.text_editor._Identity object at 0x0427F1B0>,
 'evaluate_name': '',
 'format_func': None,
 'format_str': '',
 'invalid': '',
 'is_grid_cell': False,
 'mapping': {},
 'multi_line': True,
 'password': False,
 'readonly_editor_class': <class 'enthought.traits.ui.wx.text_editor.ReadonlyEditor'>,
 'simple_editor_class': <class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 'text_editor_class': <class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 'view': None}
```

create_editor()的源代码可以在“%traits%\trait_types.py”中的 BaseStr 类的定义中找到。create_editor()得到的是一个 text_editor.ToolkitEditorFactory 类，它的完整模块路径为：

```
enthought.traits.ui.editors.text_editor.ToolkitEditorFactory
```

在“%ui%\editors\text_editor.py”中可以找到它的定义，它从 EditorFactory 继承，而 EditorFactory 类的代码在“%ui%\editor_factory.py”中。EditorFactory 类是 Trait 编辑器的核心，所有的编辑器类都通过它和后台界面库关联起来。让我们详细看看 EditorFactory 类中关于控件生成方面的代码：

```
class EditorFactory ( HasPrivateTraits ):
    # 下面 4 个属性描述 4 个类型的编辑器类
    simple_editor_class = Property ❶
    custom_editor_class = Property
    text_editor_class   = Property
    readonly_editor_class = Property

    # 用 simple_editor_class 创建实际的控件
    def simple_editor ( self, ui, object, name, description, parent ):
        return self.simple_editor_class( parent,
                                         factory     = self,
```

```
                                        ui          = ui,
                                        object      = object,
                                        name        = name,
                                        description = description )

    # 这是类的方法，它通过搜索当前类和父类，自动找到与其匹配的后台界面库中的控件类
    @classmethod
    def _get_toolkit_editor(cls, class_name): ❷
        editor_factory_classes = [factory_class for factory_class in cls.mro()
                                  if issubclass(factory_class, EditorFactory)]
        for index in range(len( editor_factory_classes )):
            try:
                factory_class = editor_factory_classes[index]
                editor_file_name = os.path.basename(
                                sys.modules[factory_class.__module__].__file__)
                return toolkit_object(':'.join([editor_file_name.split('.')[0],   ❸
                                        class_name]), True)
            except Exception, e:
                if index == len(editor_factory_classes)-1:
                    raise e
        return None

    # simple_editor_class 属性的 get 方法，获取属性值
    def _get_simple_editor_class(self):
        try:
            SimpleEditor = self._get_toolkit_editor('SimpleEditor')
        except:
            SimpleEditor = toolkit_object('editor_factory:SimpleEditor')
        return SimpleEditor
```

❶EditorFactory 对象有 4 个属性，分别用来保存后台界面库中的 4 种样式的编辑器类：simple_editor_class、custom_editor_class、text_editor_class 和 readonly_editor_class。在前面的例子中，ed 对象的 simple_editor_class 属性为：

```
>>> ed.simple_editor_class
<class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>
```

我们看到：它用的是 wx 后台界面库的 text_editor 模块中的 SimpleEditor 类，稍后将详细查看它的代码。

❷EditorFactory 类通过类方法_get_toolkit_editor()获得后台界面库中的类。由于是类方法，它的第一个参数 cls 就是 EditorFactory 类本身。当调用它的派生类的_get_toolkit_editor()时，第一个参数 cls 就是派生类本身。通过调用 cls.mro()可以获得 cls 及其所有父类，然后一个一个地

查找，从后台界面库中找到与之匹配的类，这个工作由❸toolkit_object()完成，其源代码在“%ui%\toolkit.py”中。

在后台界面库中，类的组织结构和 TraitsUI 库中的完全一样，因此不需要额外的配置文件，只需要几个字符串替代操作就可以将 TraitsUI 库中 EditorFactory 的派生类和后台界面库中实际的编辑器类关联起来。图 7-16 显示了 TraitsUI 库中 EditorFactory 和后台界面库的关系。



图 7-16　TraitsUI 库中 EditorFactory 和后台界面库的关系

后台界面库中定义了所有编辑器的控件，例如在“%wx%\text_editor.py”中可以找到产生文本框控件的类 text_editor.SimpleEditor。类名表示控件的 4 种样式——"simple"、"custom"、"text"和"readonly"，文件名(模块名)则表示控件的类型。下面是 text_editor.SimpleEditor 的部分代码：

```
class SimpleEditor ( Editor ):

    # Flag for window styles:
    base_style = 0

    # Background color when input is OK:
    ok_color = OKColor

    # Function used to evaluate textual user input:
    evaluate = evaluate_trait

    def init ( self, parent ):
        """ Finishes initializing the editor by creating the underlying toolkit
            widget.
        """
        factory       = self.factory
        style         = self.base_style
        self.evaluate = factory.evaluate
        self.sync_value( factory.evaluate_name, 'evaluate', 'from' )

        if (not factory.multi_line) or factory.password:
            style &= ~wx.TE_MULTILINE

        if factory.password:
            style |= wx.TE_PASSWORD
```

```
        multi_line = ((style & wx.TE_MULTILINE) != 0)
        if multi_line:
            self.scrollable = True

        if factory.enter_set and (not multi_line):
            control = wx.TextCtrl( parent, -1, self.str_value,
                                   style = style | wx.TE_PROCESS_ENTER )
            wx.EVT_TEXT_ENTER( parent, control.GetId(), self.update_object )
        else:
            control = wx.TextCtrl( parent, -1, self.str_value, style = style )

        wx.EVT_KILL_FOCUS( control, self.update_object )

        if factory.auto_set:
            wx.EVT_TEXT( parent, control.GetId(), self.update_object )

        self.control = control
        self.set_tooltip()
```

真正产生控件的程序在 init()方法中，此方法在产生界面时自动被调用，注意不要和对象的初始化方法__init__()搞混淆，init()在生成界面时被调用。下面查看 SimpleEditor 的父类：

```
>>> ed.simple_editor_class.mro()
[<class 'enthought.traits.ui.wx.text_editor.SimpleEditor'>,
 <class 'enthought.traits.ui.wx.editor.Editor'>,
 <class 'enthought.traits.ui.editor.Editor'>,
 <class 'enthought.traits.has_traits.HasPrivateTraits'>,
 <class 'enthought.traits.has_traits.HasTraits'>,
 <type 'CHasTraits'>,
 <type 'object'>]
```

SimpleEditor 从后台界面库的 Editor 类继承，Editor 类中定义了界面库编辑器中一些通用的属性和方法。而界面库中的 Editor 类又从 TraitsUI 库的 Editor 类继承。它定义了所有 Trait 编辑器的基本属性和功能，源代码可以在“%ui%\editor.py”中找到。

7.6.2 制作 matplotlib 的编辑器

Enthought 的官方绘图库采用的是 Chaco，不过如果读者对 matplotlib 更为熟悉，就可以在界面中使用 matplotlib 的绘图控件进行绘图。为了实现这个目的，我们需要自己编写一个 Trait 编辑器，用它封装 matplotlib 的绘图控件。下面是完整的源代码，程序的运行结果如图 7-17 所示。



mpl_figure_editor.py
嵌入 TraitsUI 界面中的 matplotlib 控件

```
import wx
import matplotlib
# matplotlib 采用 WXAgg 为后台，这样才能将绘图控件嵌入以 wx 为后台界面库的 traitsUI 窗口中
matplotlib.use("WXAgg")
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from enthought.traits.ui.wx.editor import Editor
from enthought.traits.ui.basic_editor_factory import BasicEditorFactory

class _MPLFigureEditor(Editor): ❶
    """
    相当于 wx 后台界面库中的编辑器，它负责创建真正的控件
    """
    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)
        self.set_tooltip()

    def update_editor(self): ❷
        pass

    def _create_canvas(self, parent): ❸
        """
        创建一个 Panel，布局则采用垂直排列的 BoxSizer,在 panel 中添加
        FigureCanvas、NavigationToolbar2Wx 和 StaticText 三个控件
        FigureCanvas 的鼠标移动事件调用 mousemoved 函数，在 StaticText 中
        显示鼠标所在的数据坐标
        """
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        def mousemoved(event):
            panel.info.SetLabel("%s, %s" % (event.xdata, event.ydata))
        panel.mousemoved = mousemoved
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        mpl_control = FigureCanvas(panel, -1, self.value) ❹
        mpl_control.mpl_connect("motion_notify_event", mousemoved)
        toolbar = NavigationToolbar2Wx(mpl_control)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        sizer.Add(toolbar, 0, wx.EXPAND|wx.RIGHT)
        panel.info = wx.StaticText(parent, -1)
        sizer.Add(panel.info)

        self.value.canvas.SetMinSize((10,10))
        return panel
```

```
class MPLFigureEditor(BasicEditorFactory): ❺
    """
    相当于 traits.ui 中的 EditorFactory，它返回回真正创建控件的类
    """
    klass = _MPLFigureEditor

if __name__ == "__main__":
    from matplotlib.figure import Figure
    from enthought.traits.api import HasTraits, Instance
    from enthought.traits.ui.api import View, Item
    from numpy import sin, linspace, pi

    class Test(HasTraits):
        figure = Instance(Figure, ()) ❻
        view = View(
            Item("figure", editor=MPLFigureEditor(), show_label=False),
            width = 400,
            height = 300,
            resizable = True)
        def __init__(self):
            super(Test, self).__init__()
            axes = self.figure.add_subplot(111)
            t = linspace(0, 2*pi, 200)
            axes.plot(sin(t))

    Test().configure_traits()
```

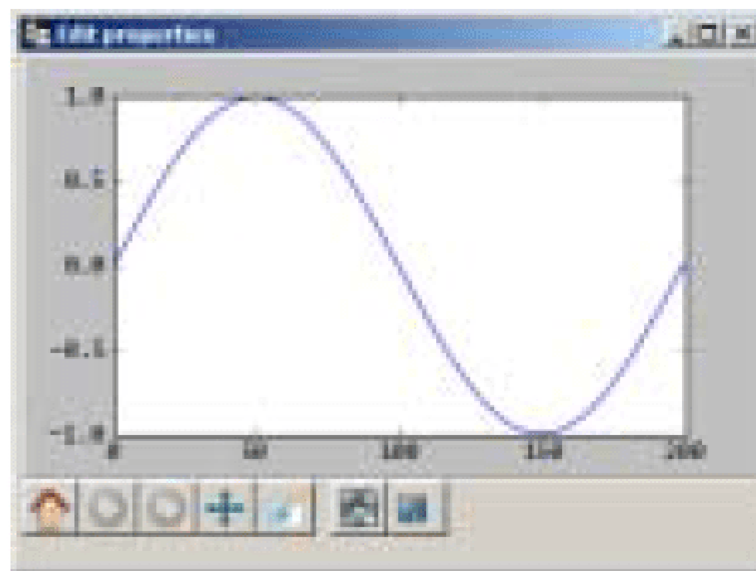


图 7-17　在 TraitsUI 界面中嵌入的 matplotlib 绘图控件

由于这个 matplotlib 绘图编辑器没有'simple'等 4 种样式，也不会放到 wx 后台界面库的模块中，因此不需要采用前面介绍的自动查找编辑器类的办法。对于这种情况，TraitsUI 提供了 BasicEditorFactory 类以方便我们实现编辑器和后台界面库的连接。它的源程序可以在“%ui%\basic_editor_factory.py”中找到。下面是其中的一部分：

```
class BasicEditorFactory ( EditorFactory ):
    klass = Any
```

```
    def _get_simple_editor_class ( self ):
        return self.klass
    ...
```

BasicEditorFactory 类通过覆盖父类的_get_simple_editor_class()方法，直接返回创建控件用的类 klass。❺MPLFigureEditors 从 BasicEditorFactory 继承，并指定创建控件的类为_MPLFigureEditor。

❶_MPLFigureEditor 和 text_editor.SimpleEditor 一样，也从 Editor 类继承，在它的 init()方法中，调用❸_create_canvas()创建实际的控件。

❷ Editor 类中有一个 update_editor()方法，它在对应的 Trait 属性改变时会被调用，因为绘图控件不需要这个功能，所以覆盖 update_editor()，让它不做任何事情。

❹在 matplotlib 中，创建 FigureCanvas 对象时需要指定与其对应的 Figure 对象，这里 self.value 就是 Figure 对象，❻它在模型类 Test 中用一个名为 figure 的 Trait 属性表示。控件可以通过其 value 属性获得模型类中它所编辑的对象。因此，_MPLFigureEditor 中的 value 属性和 Test 类中的 figure 属性是同一个 Figure 对象。

最后，_create_canvas()中的程序编写和在标准的 wx 窗口中添加控件是一样的，与界面库相关的细节不是本书的重点，因此这里不再详细解释，读者可以参考 matplotlib 和 wxPython 的相应文档。

7.6.3 CSV 数据绘图工具

用前面介绍的 matplotlib 绘图控件可以快速制作一个 CSV 数据绘图工具。用此工具打开一个 CSV 数据文档之后，可以用其中的任意两列数据为 X、Y 轴数据绘制图表。用户还可以自由地添加新的图表，修改图表的标题，选择图表的 X 轴和 Y 轴的数据。程序运行时需要从“mpl_figure_editor.py”模块载入刚才介绍的 matplotlib 绘图控件模块。程序的界面如图 7-18 所示(见文前彩插)。

traitsUI_csv_viewer.py
使用 matplotlib 绘图控件制作 CSV 数据绘图工具

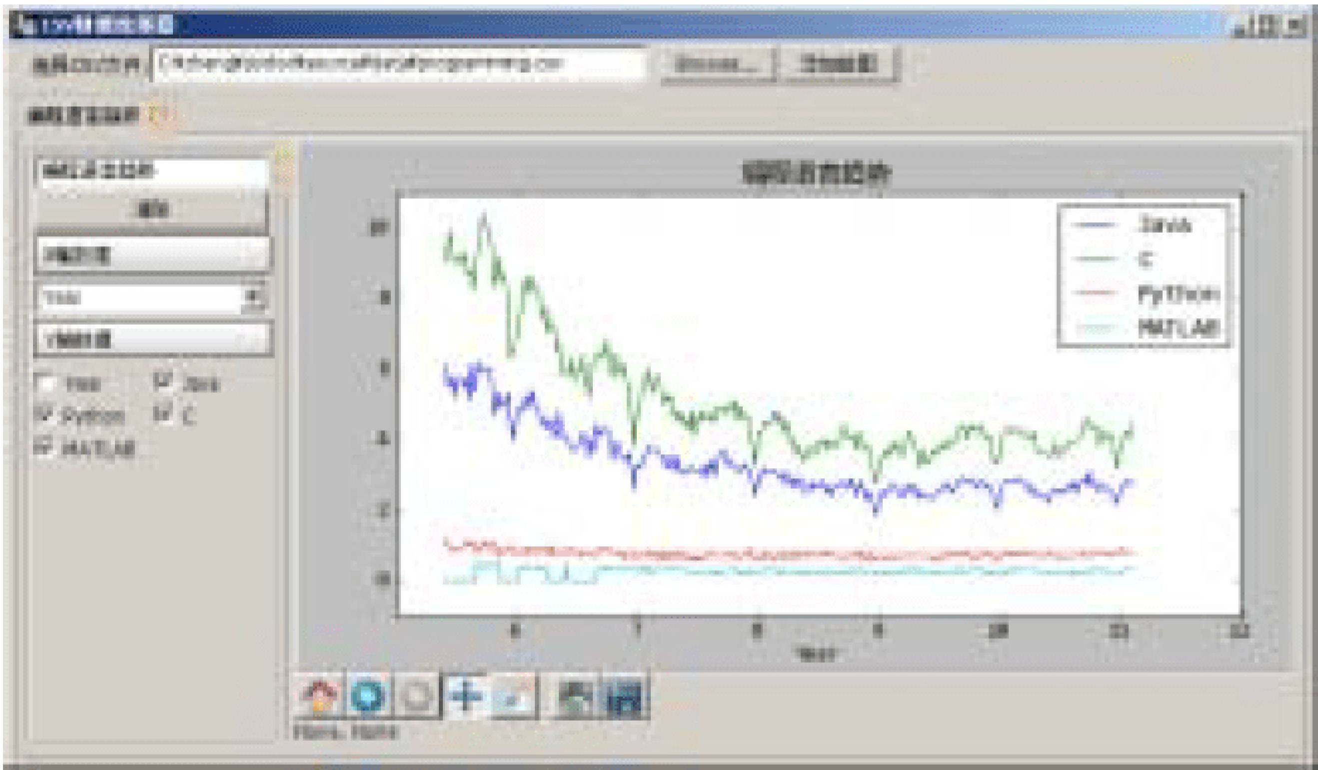


图 7-18　CSV 数据绘图工具的界面

程序以标签页的形式显示多个绘图，用户可以从左侧的数据选择栏中选择 X 轴和 Y 轴的数据。标签页可以自由地拖动，构成上下左右分栏，并且可以隐藏左侧的数据选择栏，如图 7-19 所示(见文前彩插)。

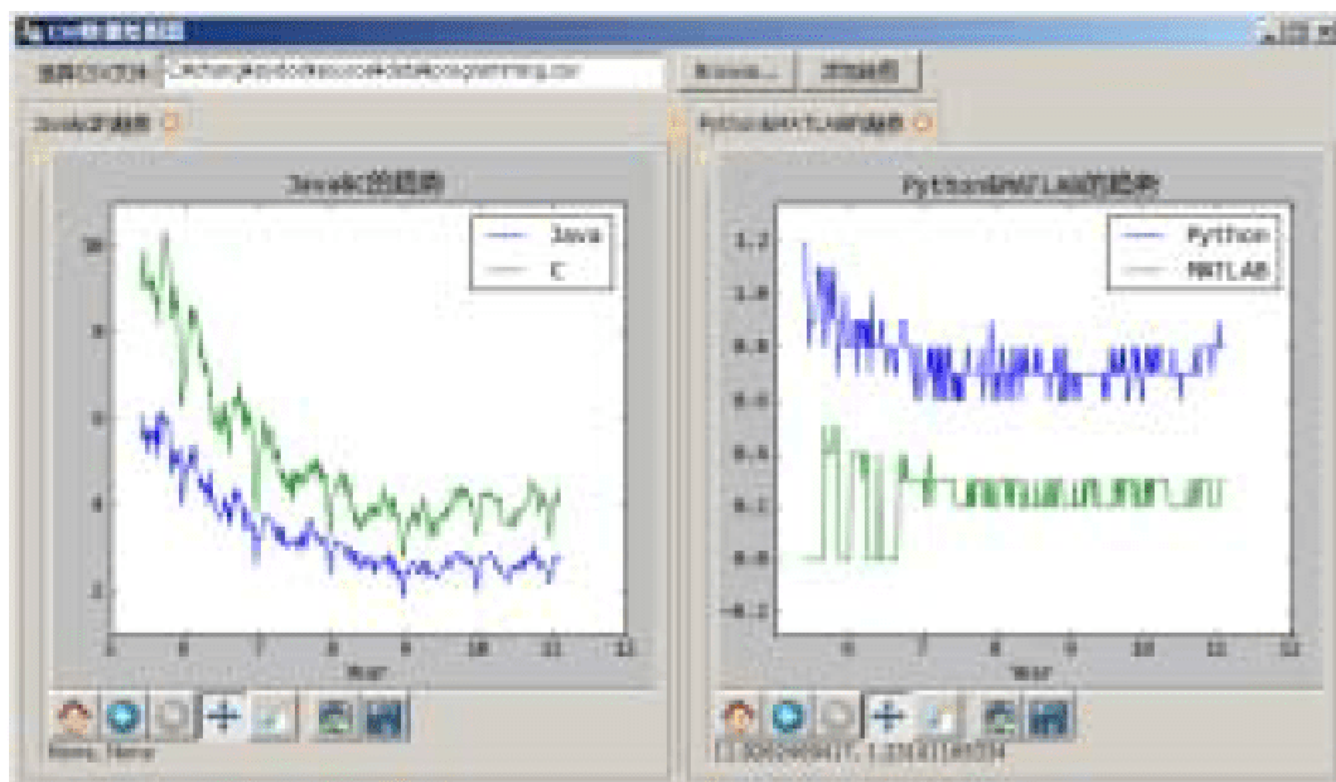


图 7-19　使用可调整 DOCK 的多标签页界面，方便用户对比数据

由于绘图控件是 matplotlib 提供的，因此平移、缩放、保存文件等功能也一应俱全。由于所有的界面都是采用 TraitsUI 设计的，因此主窗口既可以用来单独显示，也可以嵌入到一个更大的界面中，运用十分灵活。

程序中已经有比较详细的注释，这里不再重复。如果读者对 Traits 库的某项用法还不太了解，可以直接查看其源代码，代码中都有详细的注释。整个程序的界面处理都只是组装 View 对象，看不到任何关于控件操作的代码，因此大大节省了程序的开发时间。



标签页的标题如果输入中文可能会出现错误，请按照 7.4.4 节的说明修改 TraitsUI 库的源代码。

第 8 章

Chaco——交互式图表

Chaco 是 Enthought 公司开发的二维绘图库，它以 Traits 为基础，提供了十分丰富的交互功能。如果读者安装了 Python(x,y)，那么可以在它的安装目录下找到 Chaco 的演示程序：

```
c:\pythonxy\doc\Enthought Tool Suite\Chaco\examples\demo.py
```

运行此程序之后，将看到如图 8-1 所示的窗口(见文前彩插)。

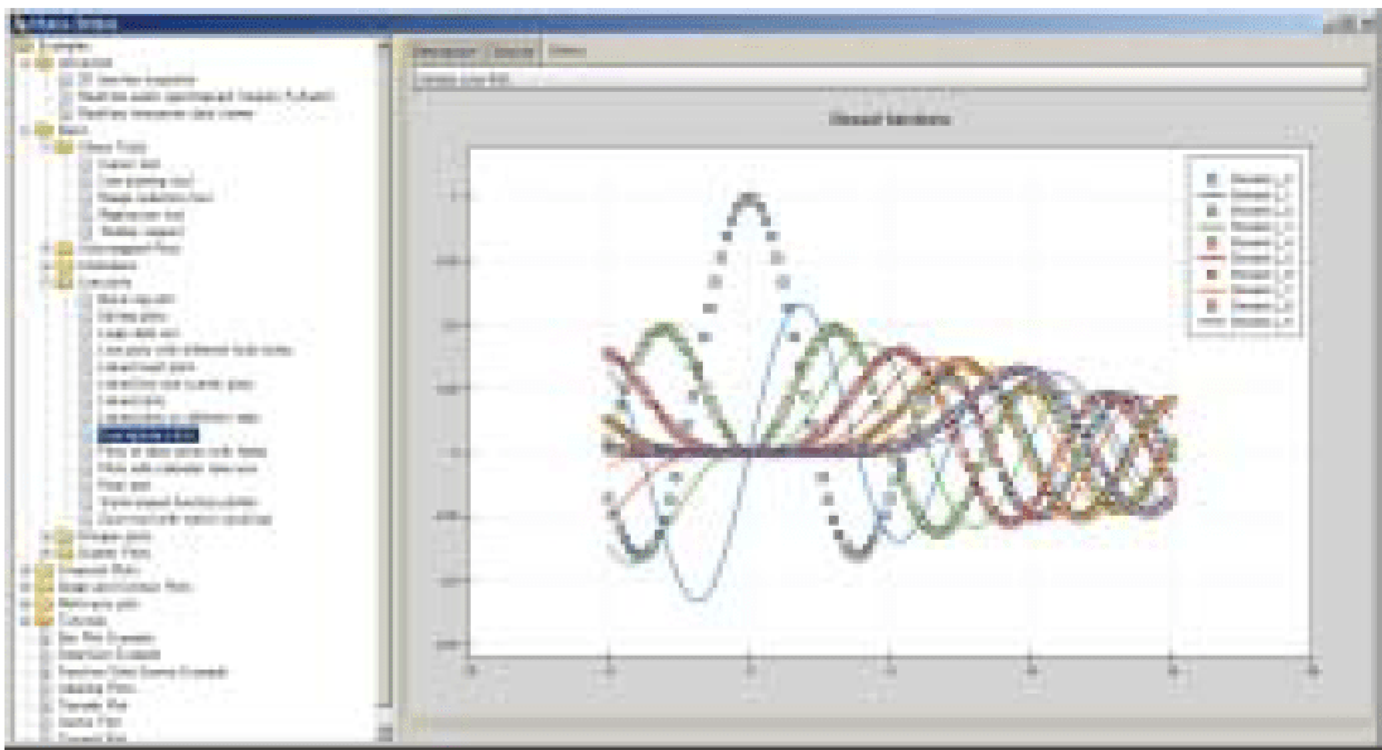


图 8-1　Chaco 的演示程序，可以同时查看源程序和运行结果

在此窗口中，选择左栏中的某个演示程序，它的相关源代码将显示在“Source”标签页下，而其运行结果则显示在“Demo”标签页下。通过学习这些演示程序，可以快速掌握 Chaco 的基本用法。

本章在简要介绍 Chaco 的基本绘图功能之后，将着重介绍各种图表交互工具的使用和开发。在应用程序的界面中使用 Chaco 图表以及各种交互工具，能提高程序的可用性，让用户更方便地观察数据。

8.1　面向脚本绘图

Chaco 提供了与 matplotlib 的 pyplot 模块类似的绘图方式，我们称之为面向脚本绘图。下面的程序使用面向脚本的方式进行快速绘图，效果如图 8-2 所示。



chaco_script.py
使用 Chaco 的面向脚本的 API 快速绘图

```
import numpy as np
import enthought.chaco.shell as cs ❶

x = np.linspace(-2*np.pi, 2*np.pi, 100)
y = np.sin(x)

cs.plot(x, y, "r-") ❷
cs.title("First plot")
cs.ytitle("sin(x)")
cs.show()
```

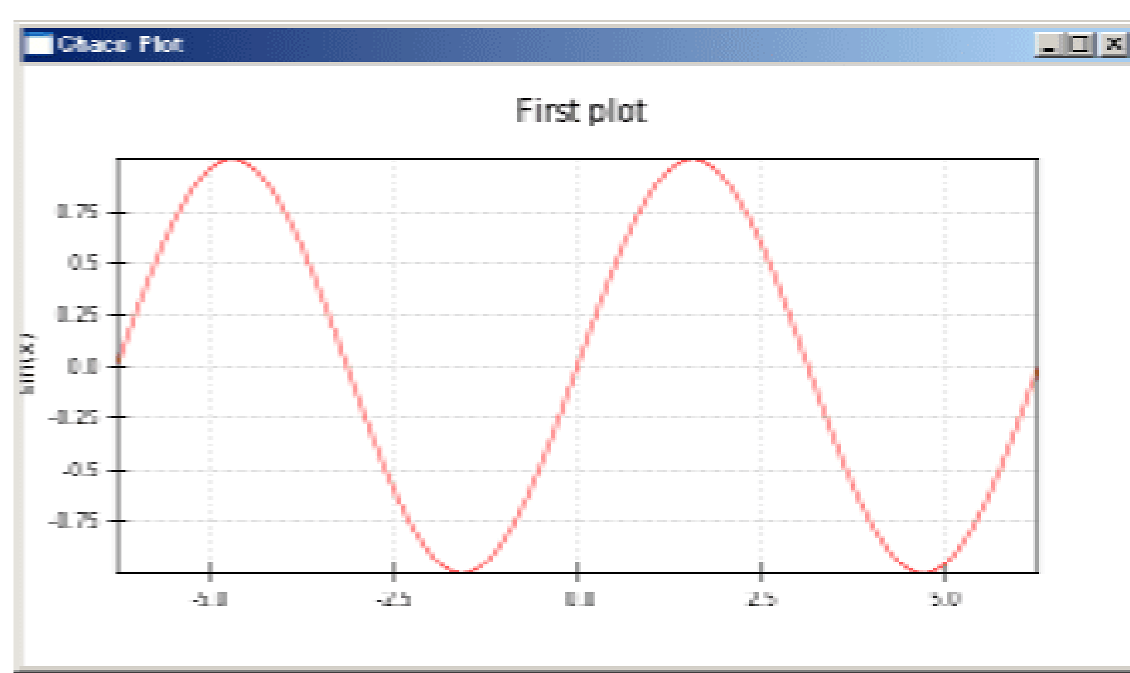


图 8-2　使用 Chaco 的脚本绘图 API 绘制正弦波

❶首先载入面向脚本的绘图模块 chaco.shell。❷调用 plot()绘制曲线，参数 x 和 y 是表示曲线上各点的 X-Y 轴坐标的数组。第三个参数指定曲线的样式，"r"表示曲线的颜色为红色，"-"表示曲线的线型为实线。接下来通过 title()为图表添加标题，ytitle()为 Y 轴添加标题，最后调用 show()显示图表。

脚本绘图不是 Chaco 的强项，虽然它的脚本绘图 API 和 matplotlib 的十分相似，但是功能却没有 matplotlib 的丰富。使用 Chaco 的优势在于它可以很方便地嵌入到用 TraitsUI 编写的界面程序中，并且提供了很多和图表进行交互的工具，能够方便我们开发包含交互式图表的应用程序。

8.2　面向应用绘图

为了将 Chaco 的图表嵌入到用 TraitsUI 制作的界面程序中，需要做一些额外的工作，因此代码量要比面向脚本绘图多许多，不过同时也更具有灵活性。让我们先从一个最简单的例子开

始，它的运行结果如图 8-3 所示，效果和采用脚本方式类似。

chaco_simple_line.py
将 Chaco 图表嵌入到用 TraitsUI 制作的界面程序中

```
import numpy as np
from enthought.traits.api import HasTraits, Instance
from enthought.traits.ui.api import View, Item
from enthought.chaco.api import Plot, ArrayPlotData ❶
from enthought.enable.component_editor import ComponentEditor ❷

class LinePlot(HasTraits):
    plot = Instance(Plot) ❸
    data = Instance(ArrayPlotData)
    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False), ❹
        width=500, height=500, resizable=True, title="Chaco Plot")

    def __init__(self, **traits):
        super(LinePlot, self).__init__(**traits)
        x = np.linspace(-14, 14, 100)
        y = np.sin(x) * x**3
        data = ArrayPlotData(x=x, y=y) ❺
        plot = Plot(data) ❻
        plot.plot(("x", "y"), type="line", color="blue") ❼
        plot.title = "sin(x) * x^3" ❽
        self.plot = plot
        self.data = data

if __name__ == "__main__":
    p = LinePlot()
    p.configure_traits()
```

程序看起来比较复杂，但只要掌握了它的基本设计思想，就很容易理解了。程序中首先从许多模块载入对象，❶从 Chaco 库载入 Plot 类和 ArrayPlotData 类，Plot 是一个封装了各种绘图功能的类，而 ArrayPlotData 是用来保存绘图数据的类。即 Plot 负责图表的绘制，ArrayPlotData 负责管理图表数据。❷从 Enable 库载入 ComponentEditor，它负责在界面上显示

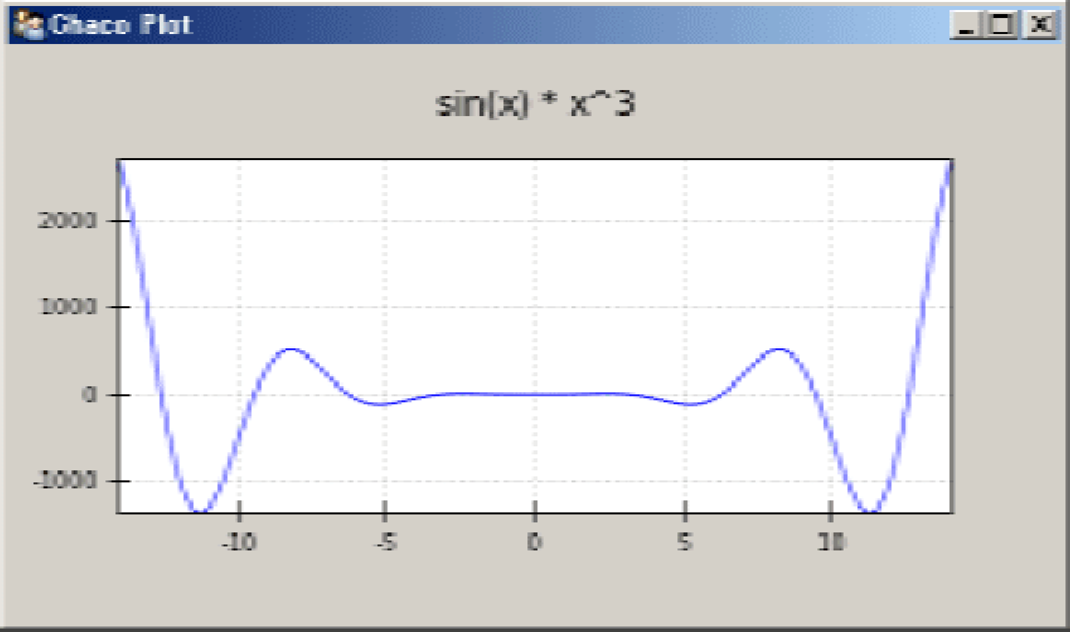


图 8-3　将 Chaco 图表嵌入到使用 TraitsUI 制作的界面程序中

Plot 对象所表示的图表。

LinePlot 类从 HasTraits 继承，❸它有两个 Trait 属性——plot 和 data，用 Instance 定义它们分别是 Plot 和 ArrayPlotData 的实例。❹在视图的定义中，指定 plot 属性的编辑器为一个 ComponentEditor 对象，这样它就能负责在界面中显示 Plot 对象了。

在 LinePlot 类的初始化方法__init__()中，首先计算出表示正弦波的数组 x 和 y。❺然后创建一个存储绘图数据的 ArrayPlotData 对象 data，并通过关键字参数将两个数组传递给它。ArrayPlotData 对象可以像字典一样使用，可以通过关键字参数名获得其对应的数组。例如，在 IPython 命令行中输入下面的语句，可以从 data 对象获得数组 x：

```
>>> run chaco_simple_line.py
>>> p.data["x"]
array([-14.        , -13.71717172, -13.43434343, -13.15151515,……
```

❻接下来创建 Plot 对象，并将存储绘图数据的 ArrayPlotData 对象传递给它，此后 Plot 对象将通过数组名从 ArrayPlotData 对象中获得实际的数组。这样就在图表和数据之间形成了一个接口：修改 ArrayPlotData 对象中的数组将会反映到与此数组相关的 Plot 对象之上，而多个 Plot 对象也可以共用同一个 ArrayPlotData 对象。通过 Plot 对象的 data 属性可以获得与其关联的 ArrayPlotData 对象：

```
>>> p.plot.data == p.data
True
```

Plot 类对 Chaco 库中的许多绘图类进行封装，提供了一个统一的接口以方便用户创建和管理这些绘图对象。❼调用其 plot()创建一个曲线图对象，我们只将数组名传递给它，它将通过数组名在其 data 属性中找到对应的数组。type 参数为"line"表示绘制曲线图，曲线的颜色由 color 参数指定。❽最后通过设置 Plot 对象的 title 属性，修改图表的标题。

显然，这段程序比前面的脚本绘图程序要复杂许多，为了方便读者理解，下面对上面的绘图程序做一个总结：

- 创建一个从 HasTraits 继承的类，给它定义一个类型为 Plot 的 Trait 属性，并在视图定义中为此属性指定一个 ComponentEditor 对象作为编辑器。
- 创建表示绘图数据的数组，并使用 ArrayPlotData 对象封装这些数组。
- 创建 Plot 对象，将 ArrayPlotData 对象传递给它，并用 plot 属性保存 Plot 对象。
- 调用 Plot 对象的 plot()方法绘图，不直接将数组传递给 plot()，而是使用 ArrayPlotData 对象中的数组名。

8.2.1 多条曲线

采用和上一节相同的方法，我们可以用下面的程序在一幅图表中绘制多条曲线，效果如图 8-4 所示(见文前彩插)。由于程序的结构和上一节的例子相同，下面只给出初始化方法__init__()的代码：



chaco_multiple_line.py
绘制多条曲线

```
def __init__(self, **traits):
    super(MutiLinePlot, self).__init__(**traits)
    x = np.linspace(-14, 14, 100)
    y1 = np.sin(x) * x**3
    y2 = np.cos(x) * x**3
    data = ArrayPlotData(x=x, y1=y1, y2=y2)
    plot = Plot(data)
    plot.plot(("x", "y1"), type="line", color="blue", name="sin(x) * x**3") ❶
    plot.plot(("x", "y2"), type="line", color="red", name="cos(x) * x**3")
    plot.plot(("x", "y2"), type="scatter", color="red", marker = "circle", ❷
              marker_size = 2, name="cos(x) * x**3 points")
    plot.title = "Multiple Curves"
    plot.legend.visible = True ❸
    self.plot = plot
    self.data = data
```

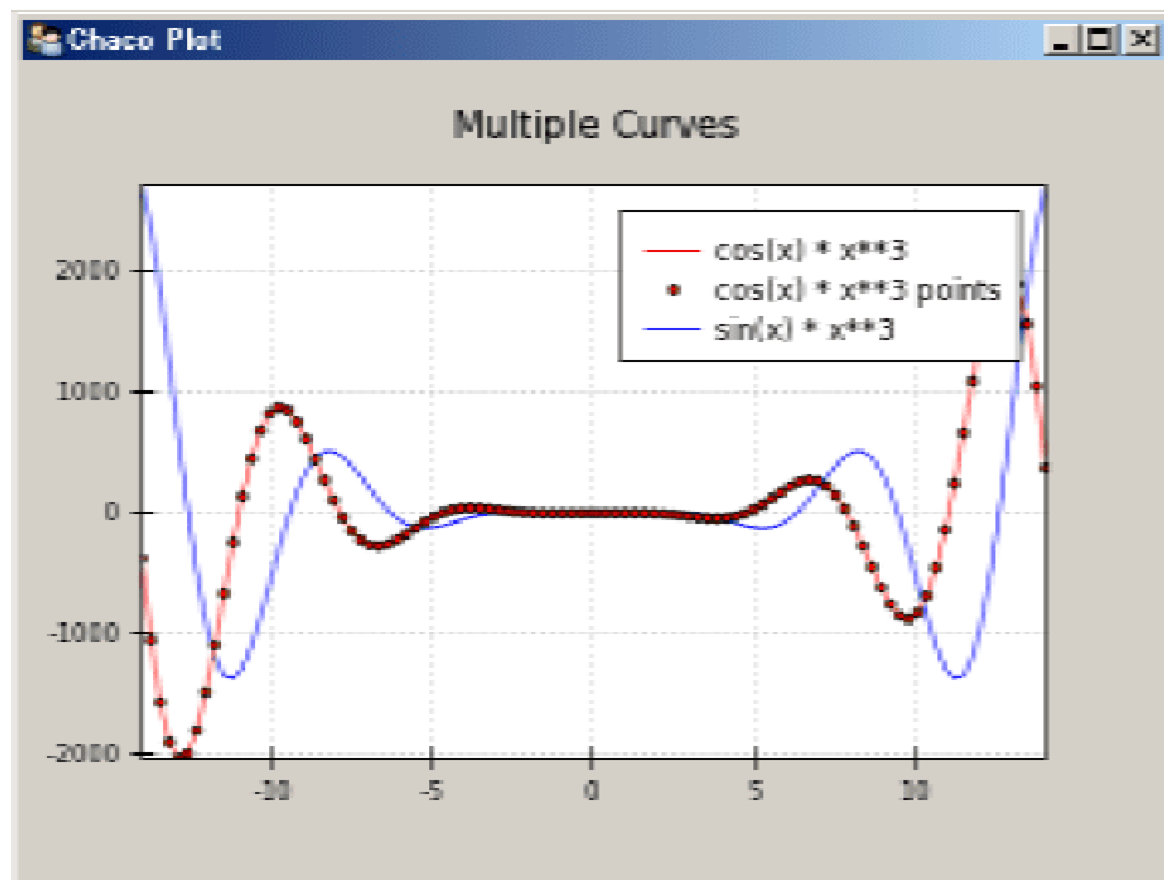


图 8-4　绘制多条曲线并显示图示

在__init__()中调用了 3 次 Plot 对象的 plot()方法，❶前两次分别绘制正弦曲线和余弦曲线，type 参数为"line"，表示绘制曲线图。❷type 参数为"scatter"，表示只绘制数据点，不使用直线将它们连接起来。通过 marker 和 marker_size 等参数可以设置点的样式和大小。

❸Plot 对象的 legend 属性是表示图示的对象，这里通过设置它的 visible 属性在图表上显示图示。图示中将显示每次调用 plot()时传递的 name 参数。如果没有设置 name 参数，系统会自动为其设置一个值。

下面在 IPython 中运行程序，然后观察 Plot 对象的内容：

```
>>> run chaco_multiple_line.py
>>> plot = p.plot # p 是包含 Plot 对象的 MutiLinePlot 对象
```

plot 是一个 Plot 对象，让我们看看 Plot 类的继承列表：

```
>>> plot.__class__.mro()
[<class 'enthought.chaco.plot.Plot'>,
 <class 'enthought.chaco.data_view.DataView'>,
 <class 'enthought.chaco.plot_containers.OverlayPlotContainer'>,
 <class 'enthought.chaco.base_plot_container.BasePlotContainer'>,
 <class 'enthought.enable.container.Container'>,
 <class 'enthought.enable.component.Component'>,
 <class 'enthought.enable.coordinate_box.CoordinateBox'>,
 <class 'enthought.enable.interactor.Interactor'>,
 <class 'enthought.traits.has_traits.HasTraits'>,
 <type 'CHasTraits'>,
 <type 'object'>]
```

由于 Plot 从 HasTraits 继承，因此它的所有属性都是 Trait 属性。它从 Enable 库的 Component 类(组件类)继承，所以在定义视图时可以使用 Enable 库的 ComponentEditor 作为 Plot 对象的编辑器，关于这部分内容将在后面详细介绍。它还从 Container 继承，因此 Plot 对象可以包含其他的绘图对象。在上面的例子中，Plot 对象包含了通过 plot()方法创建的三条曲线(其中有一条只显示数据点)，这些绘图对象都保存在 Plot 对象的 components 属性中。



每个 Trait 属性的用法在源程序中都有很详细的注释，因此建议读者通过阅读 Chaco 的源程序来加深理解。

```
>>> plot.components
[<enthought.chaco.lineplot.LinePlot object at 0x05A35C00>,
 <enthought.chaco.lineplot.LinePlot object at 0x05A3C060>,
 <enthought.chaco.scatterplot.ScatterPlot object at 0x05A3C3F0>]
```

我们看到：当 type 参数为"line"时，plot()创建 LinePlot 对象表示曲线图；当 type 参数为"scatter"时，创建 ScatterPlot 对象表示散列图。

Plot 对象的 plots 属性是一个包含用 plot()创建的绘图对象的字典，键是绘图对象的 name 属性：

```
>>> plot.plots
{'cos(x) * x**3': [<enthought.chaco.lineplot.LinePlot object at 0x05A3C060>],
...
```

Plot 对象的 overlays 属性中有一个 PlotLabel 对象和一个 Legend 对象，它们分别是图表的标题和图示：

```
>>> plot.overlays
[<enthought.chaco.plot_label.PlotLabel object at 0x05A30E70>,
 <enthought.chaco.legend.Legend object at 0x05A35810>]
```

underlays 属性中保存有两个 PlotGrid 对象和两个 PlotAxis 对象，它们分别是图表中 X 轴和 Y 轴的网格，以及 X 轴和 Y 轴本身：

```
>>> plot.underlays
[<enthought.chaco.grid.PlotGrid object at 0x05A2A510>,
 <enthought.chaco.grid.PlotGrid object at 0x05A2AED0>,
 <enthought.chaco.axis.PlotAxis object at 0x05A308A0>,
 <enthought.chaco.axis.PlotAxis object at 0x05A30B70>]
```

所有这些对象都从 HasTraits 继承，可以调用各个对象的 edit_traits()方法，显示出编辑它们的属性的界面，如图 8-5 所示，通过这些界面可以交互式地修改对象的各种属性。



有些对象没有默认视图，因此调用它们的 edit_traits()方法会自动根据其 Trait 属性创建界面，而自动创建界面可能会失败。这时我们可以在调用 edit_traits()时传递一个自己创建的视图对象。

```
>>> plot.underlays[0].edit_traits()
>>> plot.underlays[2].edit_traits()
>>> plot.components[0].edit_traits()
```

Plot 对象会按照一定的顺序绘制其中包含的各个对象。这个顺序可以通过其 draw_order 属性获得：

```
>>> p.plot.draw_order
['background', 'image', 'underlay', 'plot',
 'selection', 'border', 'annotation', 'overlay']
```

可以将 draw_order 属性中的每个元素当做一个类似于透明纸的绘图层，Plot 对象中的每个绘图元素都在其中的一张透明纸上进行绘制，最终的结果就是这些透明纸按照 draw_order 的顺序从下往上叠加在一起。underlays 属性中的对象在'underlay'层上绘制，components 属性中的对象在'plot'层上绘制，而 overlays 属性中的对象则在'overlay'层上绘制。因此坐标轴和网格被数据曲线覆盖，而标题和图示则能够覆盖数据曲线。

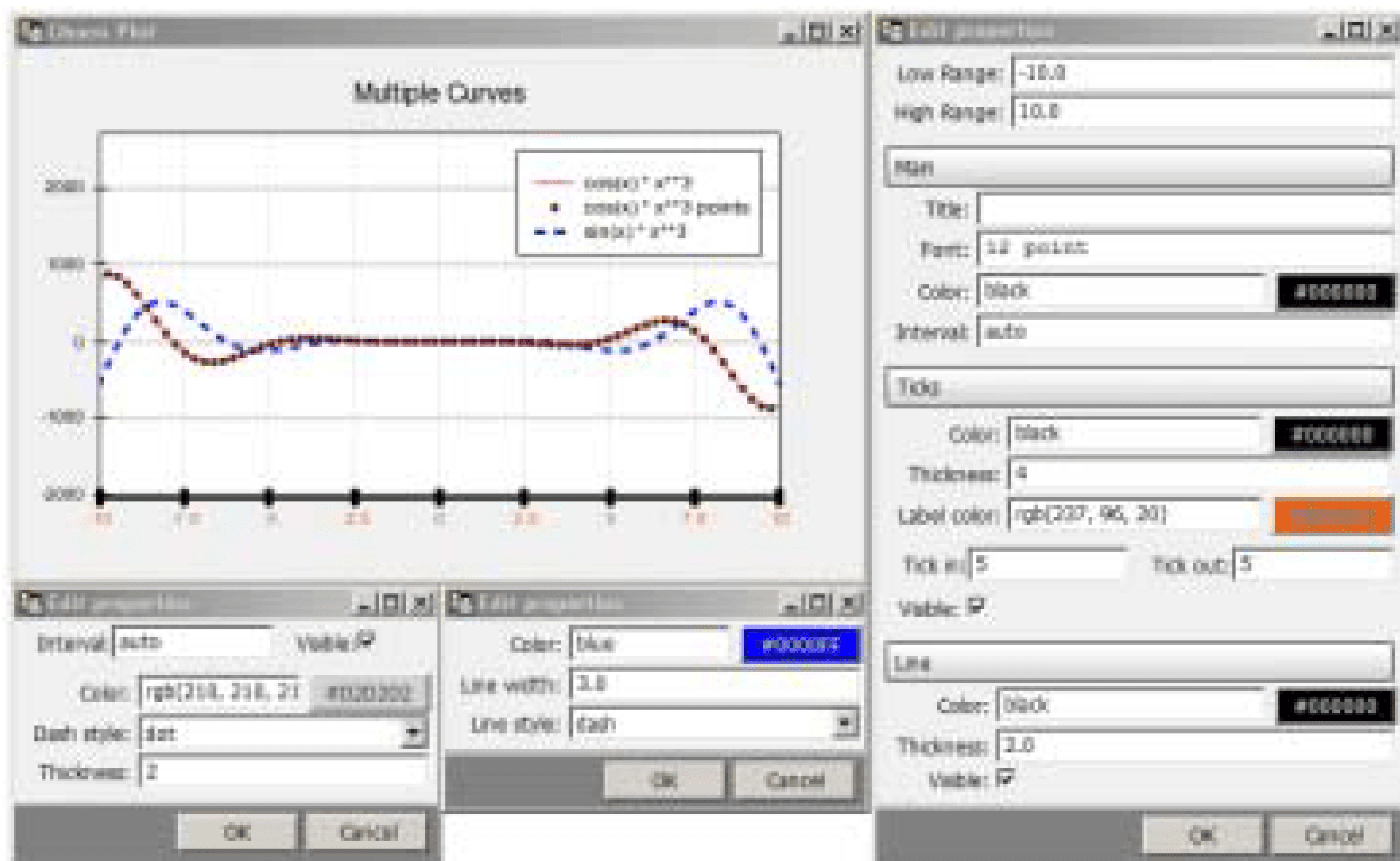


图 8-5　用 edit_traits()显示出各个对象的编辑界面

LinePlot、ScatterPlot、PlotLabel、Legend、PlotGrid 和 PlotAxis 等都是绘图组件，它们都从 Enable 库的 Component 继承，例如下面的程序查看 PlotLabel 的父类：

```
>>> plot.overlays[0].__class__.mro()
[<class 'enthought.chaco.plot_label.PlotLabel'>,
 <class 'enthought.chaco.abstract_overlay.AbstractOverlay'>,
 <class 'enthought.chaco.plot_component.PlotComponent'>,
 <class 'enthought.enable.component.Component'>,
 <class 'enthought.enable.coordinate_box.CoordinateBox'>,
 <class 'enthought.enable.interactor.Interactor'>,
 <class 'enthought.traits.has_traits.HasTraits'>,
 <type 'CHasTraits'>,
 <type 'object'>]
```

和 Plot 类不同的是，PlotLabel 不从 Container 继承，因此它不能包含其他的绘图元素。我们可以做如下总结：图表中的每个组件都是一个 Component 对象，有些组件还是 Container 对象，它们可以包含其他的组件，有些不是 Container 对象，它们与图表中的某个绘图实体对应。

8.2.2　Plot 对象的结构

前面介绍了 Plot 对象有三个属性分别保存它所包含的其他组件：overlays、underlays 和 components。这些属性都和绘图相关，Plot 类从 Enable 库的 Component 类和 Container 类继承了这些属性。而一幅图表除了能显示绘图结果之外，还需要能对数据进行处理，因此 Plot 类中还有许多和数据相关的属性。图 8-6 显示了 Plot 对象和其他对象之间的关系。下面以上一节的

“chaco_multiple_line.py”为例，在 IPython 中观察 Plot 对象的内部结构。

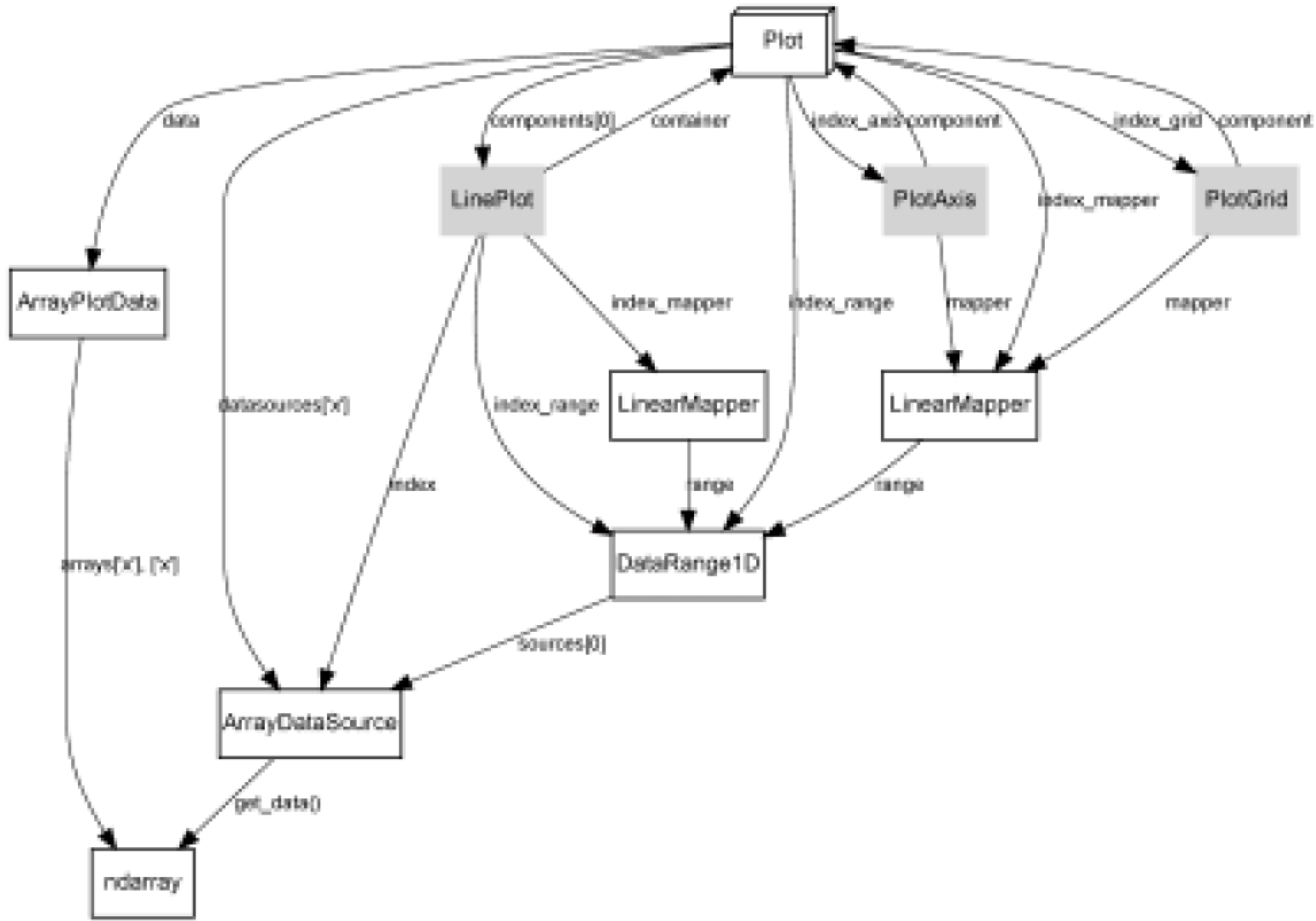


图 8-6　Plot 对象的结构

> 本节介绍的类大都从 HasTraits 继承，因此可以通过它们的 get()或 print_traits()方法快速查看所有的 Trait 属性。

Plot 对象是一个容器，图中用立体矩形表示。LinePlot、PlotAxis 和 PlotGrid 是绘图组件，用填充矩形表示。图中的箭头表示对象之间的引用关系。例如 Plot 对象的 data 属性是一个 ArrayPlotData 对象，因此图中用一条标记为 data 的箭头曲线将二者连接起来：

```
>>> plot = p.plot
>>> plot.data
<enthought.chaco.array_plot_data.ArrayPlotData object at 0x0524E270>
```

ArrayPlotData 的 array 属性是一个字典，它将数组名和数组联系起来。ArrayPlotData 同时覆盖了__getitem__()，因此它本身也可以当做字典使用：

```
>>> plot.data["x"]
array([-14.        , -13.71717172, -13.43434343, ...])
```

Plot 对象同时提供了一个用于保存数据的 datasources 属性，它和字典对象类似，键是数组

名，值是对数组进行封装的 ArrayDataSource 对象。通过 ArrayDataSource 对象的 get_data()方法可以获得其内部封装的数组。ArrayDataSource 对象还提供了一个用来描述数组的 metadata 字典，可以往其中任意添加描述数组的数据。在后面介绍交互工具时，我们还会详细学习 metadata 字典的用法。

```
>>> plot.datasources["x"]
<enthought.chaco.array_data_source.ArrayDataSource object at 0x052DB900>
>>> plot.datasources["x"].get_data()
array([-14.        , -13.71717172, -13.43434343, ...])
>>> plot.datasources["x"].metadata
{'annotations': [], 'selections': []}
```

Plot 对象是一个容器组件，它所包含的组件都保存在 components 列表属性中。调用 plot()时，会自动创建各种绘图组件并将它们添加到此列表中。在本例中，第一次调用 plot()时创建的是一个 LinePlot 对象，通过它的 container 属性可以获得包含它的 Plot 对象。LinePlot 对象的 index 属性表示曲线 X 轴的数据，value 属性表示 Y 轴的数据。它们都是 Plot 对象的 datasources 属性中的 ArrayDataSource 对象。

```
>>> plot.components[0]
<enthought.chaco.lineplot.LinePlot object at 0x052DB9F0>
>>> plot.components[0].index
<enthought.chaco.array_data_source.ArrayDataSource object at 0x052DB900>
```

LinePlot 对象使用两个 LinearMapper 对象——index_mapper 和 value_mapper，分别用来将 X 轴和 Y 轴的数据转换到屏幕坐标系。

```
>>> plot.components[0].index_mapper
<enthought.chaco.linear_mapper.LinearMapper object at 0x052DB930>
```

在 LinearMapper 对象中，map_screen()将数组从数据坐标系转换为屏幕坐标系。map_data()是 map_screen()的逆变换，它将数组从屏幕坐标系转换为数据坐标系。

```
>>> plot.components[0].index_mapper.map_screen(np.array([0,1,2]))
Out[23]: array([ 190.5       ,  204.10714286,  217.71428571])
```

为了进行坐标变换，LinearMapper 对象通过它的 range 属性获得数据的范围，range 属性是一个 DataRange1D 对象。由于 DataRange1D 对象可以同时计算多组数据的范围，因此它的 sources 属性是一个 ArrayDataSource 对象的列表。

```
>>> plot.components[0].index_mapper.range
<enthought.chaco.data_range_1d.DataRange1D object at 0x052D10F0>
>>> plot.components[0].index_mapper.range.sources[0]
Out[29]: <enthought.chaco.array_data_source.ArrayDataSource object at 0x052DB900>
```

通过 LinePlot 对象的 index_range 属性也可以获得表示 X 轴数据范围的 DataRange1D 对象，通过 value_range 属性获得表示 Y 轴数据范围的对象。

```
>>> plot.components[0].index_range
<enthought.chaco.data_range_1d.DataRange1D object at 0x052D10F0>
```

Plot 对象的 underlays 列表中包含两个 PlotAxis 对象和两个 PlotGrid 对象，它们也可以通过 Plot 对象的 index_axis、value_axis、index_grid 和 value_grid 等属性获得。通过它们的 component 属性可以获得包含它们的 Plot 对象。PlotAxis 和 PlotGrid 对象也需要进行坐标系变换运算，因此它们的 mapper 属性都指向一个 LinearMapper 对象。注意它和 LinePlot 对象的 index_mapper 不是同一个对象。但是这两个 LinearMapper 对象使用同一个 DataRange1D 对象。这部分内容请读者自行在 IPython 中进行验证。

Chaco 提供多种进行坐标转换的 Mapper 类，多种表示数据范围的 DataRange 类，以及多种表示数据源的 DataSource 类。它们分别从 AbsractMapper、AbstractDataRange 和 AbstractDataSource 等抽象类继承。下面的小程序输出这些抽象类的所有子类。



chaco_classes.py
输出各个抽象类的子类

程序的输出如下：

```
<class 'enthought.chaco.abstract_mapper.AbstractMapper'>
<class 'enthought.chaco.base_1d_mapper.Base1DMapper'>
<class 'enthought.chaco.linear_mapper.LinearMapper'>
<class 'enthought.chaco.log_mapper.LogMapper'>
<class 'enthought.chaco.grid_mapper.GridMapper'>
<class 'enthought.chaco.polar_mapper.PolarMapper'>

<class 'enthought.chaco.abstract_data_range.AbstractDataRange'>
<class 'enthought.chaco.base_data_range.BaseDataRange'>
<class 'enthought.chaco.data_range_1d.DataRange1D'>
<class 'enthought.chaco.data_range_2d.DataRange2D'>

<class 'enthought.chaco.abstract_data_source.AbstractDataSource'>
<class 'enthought.chaco.array_data_source.ArrayDataSource'>
<class 'enthought.chaco.point_data_source.PointDataSource'>
<class 'enthought.chaco.grid_data_source.GridDataSource'>
<class 'enthought.chaco.image_data.ImageData'>
<class 'enthought.chaco.multi_array_data_source.MultiArrayDataSource'>
```

在每个类的源程序中都有详细的使用说明，相信读者在掌握了本节介绍的结构关系之后，很容易通过阅读程序中的文档掌握它们的用法。

8.2.3 编辑绘图属性

绘图组件都从 HasTrait 类继承，因此它们的属性都是 Trait 属性，我们可以使用 TraitsUI 为这些属性在界面中生成编辑控件。下面的程序演示了这一过程，效果如图 8-7 所示。



chaco_app_example.py
使用 TraitsUI 界面编辑组件的属性

```
from enthought.chaco.api import marker_trait
class ScatterPlotTraits(HasTraits):
    plot = Instance(Plot)
    data = Instance(ArrayPlotData)
    color = Color("blue")
    marker = marker_trait   ❶

    traits_view = View(
        Group(Item('color', label="Color"),
              Item('marker', label="Marker"),
              Item('object.line.marker_size', label="Size"), ❷
              Item('plot', editor=ComponentEditor(), show_label=False),
                  orientation = "vertical"),
              width=800, height=600, resizable=True, title="Chaco Plot")

    def __init__(self, **traits):
        super(ScatterPlotTraits, self).__init__(**traits)
        x = np.linspace(-14, 14, 100)
        y = np.sin(x) * x**3
        data = ArrayPlotData(x = x, y = y)
        plot = Plot(data)

        self.line = plot.plot(("x", "y"), type="scatter", color="blue")[0] ❸
        self.plot = plot
        self.data = data

    def _color_changed(self):
        self.line.color = self.color

    def _marker_changed(self):
        self.line.marker = self.marker

if __name__ == "__main__":
    p = ScatterPlotTraits()
    p.configure_traits()
```

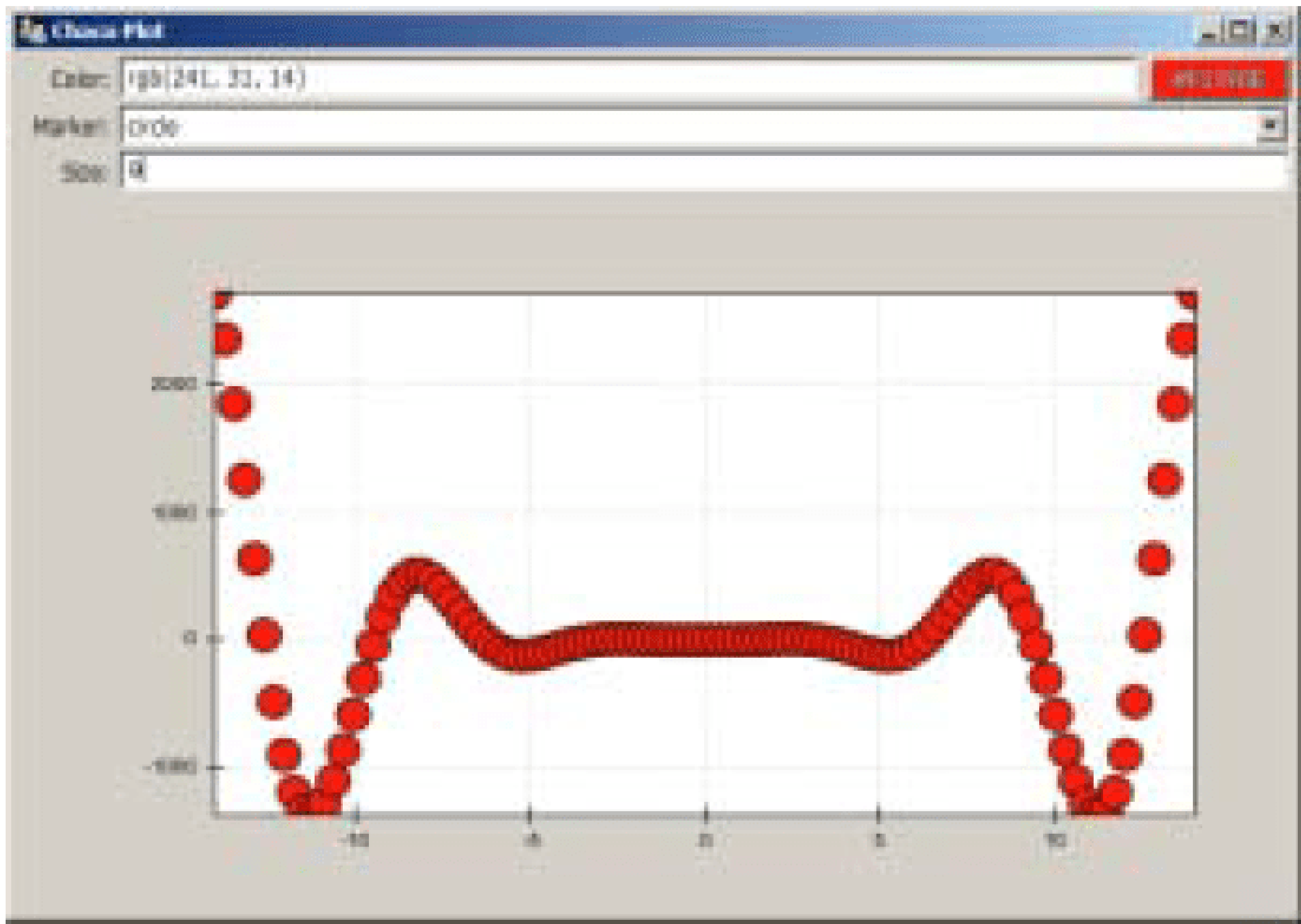


图 8-7　带属性编辑器的 Chaco 绘图界面

ScatterPlotTraits 类中定义了 4 个 Trait 属性，其中两个是我们已经熟知的 plot 和 data 属性，而 color 和 marker 属性分别用来修改曲线的颜色和散列点的形状。❶marker 属性的类型为 marker_trait，marker_trait 是在 Chaco 库中定义的一个 Trait 类型，它将散列点的形状名映射到与其对应的类。我们为这些属性定义了事件处理方法，当它们的值改变时会调用_color_changed()和_marker_changed()。在这两个事件处理方法中通过修改 line 的属性设置曲线的绘图属性。

❸line 属性保存 plot()所返回的列表中的第一个元素，也就是与图中曲线相对应的 LinePlot 对象。也可以不保存返回值，直接使用前面介绍的 components 或 plots 属性来获得它。例如可以用“plot.plots["plot0"]”获取此 LinePlot 对象。"plot0"是系统自动为曲线所起的名字。注意：在 ScatterPlotTraits 类中并没有预先定义 line 属性，它是在__init__()中动态添加的 Trait 属性。

❷也可以直接在界面中为 LinePlot 对象的属性创建编辑控件，省去先在 ScatterPlotTraits 类中定义 Trait 属性，然后进行事件处理的步骤。"object"代表拥有此视图的 ScatterPlotTraits 对象，"line"是用来保存 LinePlot 对象的属性名，而"marker_size"是 LinePlot 对象的属性名，用来设置散列点的大小。

8.2.4　容器(Container)

在 Plot 的父类列表中有 OverlayPlotContainer 类，它将 components 属性中的所有组件重叠在一起进行显示，因此 Plot 对象可以显示多条曲线。除此之外，在 Chaco 库中还提供了几种容器，完整的容器列表如下：

- OverlayPlotContainer：将多个组件进行重叠显示的容器。
- HPlotContainer：横向排列的容器。
- VPlotContainer：竖向排列的容器。
- GridPlotContainer：按照网格排列的容器。

下面的例子使用各种容器创建一个布局比较复杂的多子图图表，效果如图 8-8 所示。程序和前面的例子类似，因此我们只分析其中创建图表部分的代码。



chaco_containers.py
使用各种容器制作多子图图表

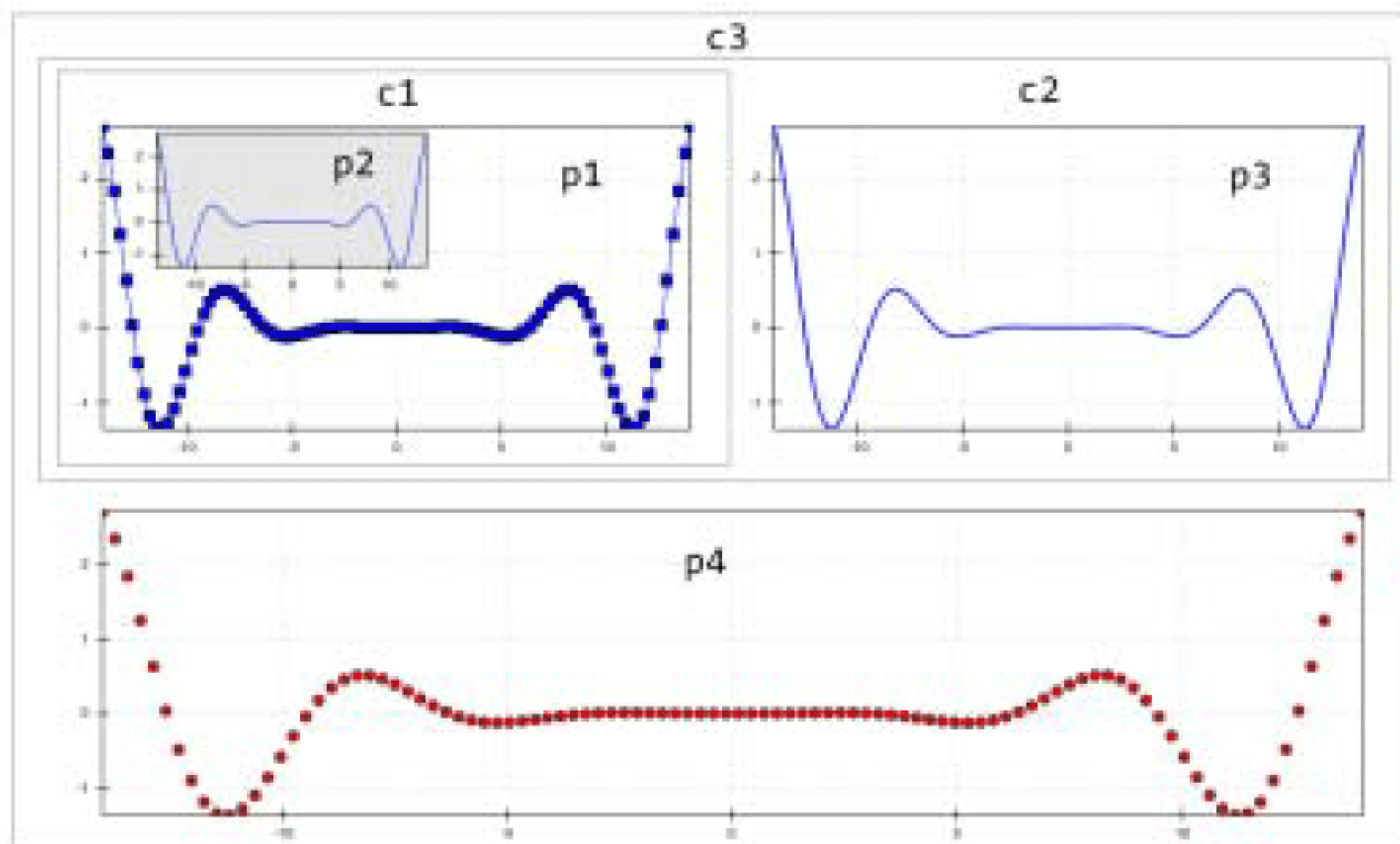


图 8-8　使用各种容器制作的多子图图表

```
p1 = Plot(data, padding=30)
p1.plot(("x", "y"), type="scatter", color="blue")
p1.plot(("x", "y"), type="line", color="blue")
p2 = Plot(data, padding=30)
p2.plot(("x", "y"), type="line", color="blue")
p2.set(bounds = [200, 100], position = [70,150],  ❶
bgcolor = (0.9,0.9,0.9), unified_draw=True, resizable="")
p3 = Plot(data, padding=30)
p3.plot(("x", "y"), type="line", color="blue", line_width=2.0)
p4 = Plot(data,  padding=30)
p4.plot(("x", "y"), type="scatter", color="red", marker="circle")
c1 = OverlayPlotContainer(p1, p2) ❷
c1.fixed_preferred_size = p3.get_preferred_size() ❸
c2 = HPlotContainer(c1, p3) ❹
c3 = VPlotContainer(p4, c2) ❺
self.plot = c3
```

首先，p1、p2、p3 和 p4 是 4 个 Plot 对象。c1 是一个 OverlayPlotContainer 容器，c2 是一个 HPlotContainer 容器，c3 是一个 VPlotContainer 容器。各个 Plot 对象和容器所对应的显示区域已经在图 8-8 中标出。每个组件对象都有一个 padding 属性设置其边距，为了让子图的排列更紧凑一些，我们在创建 Plot 对象时设置它们的边距为 30 个像素。也可以使用 padding_left、padding_right、padding_top 和 padding_bottom 属性分别设置左右上下 4 个边距。

❷使用重叠容器 c1 将 p1 和 p2 两个 Plot 对象重叠显示。❶通过设置 p2 对象的 position 属性和 bounds 属性，指定它在容器 c1 中的显示区域。为了让 p2 不自动调节自己的大小，还需要设置其 resizable 属性为空字符串。此外，通过 bgcolor 属性指定 p2 的背景颜色，并指定 unified_draw 属性为 True。绘图对象 p2 的 unified_draw 属性为 True，表示它将作为一个整体绘制到容器 c1 的一个绘图层(默认为'plot'绘图层)上，否则它的不同绘图层将分别绘制到容器 c1 对应的绘图层上。所有这些属性都是 Trait 属性，因此可以通过 set()设置它们。



读者可以将 p2 的 unified_draw 属性修改为 False，通过观察绘图结果的变化，理解 unified_draw 属性以及绘图层的含义。

❹使用横排容器 c2 将容器 c1 和图表 p3 横向排列，容器会根据各个组件的尺寸自动调节它们的大小比例。❸为了让 c1 和 p3 一样大，通过 get_preferred_size()获得 p3 的首选尺寸，并赋值给 c1 的 fixed_preferred_size 属性。这样一来，c1 和 p3 的首选尺寸相同，容器 c2 就能让它们的尺寸始终保持一致。

❺最后用竖排容器 c3 将 p4 和 c2 竖向排列，由于坐标原点在左下角，因此 p4 在 c2 的下方。

图 8-9 显示了图 8-8 中各个部分之间的关系。图中，灰色填充矩形表示容器，而无填充矩形表示图表上的各种绘图组件，虚线矩形表示列表。图中只显示了一个 Plot 对象的内部构成，其他的 Plot 对象和它类似。

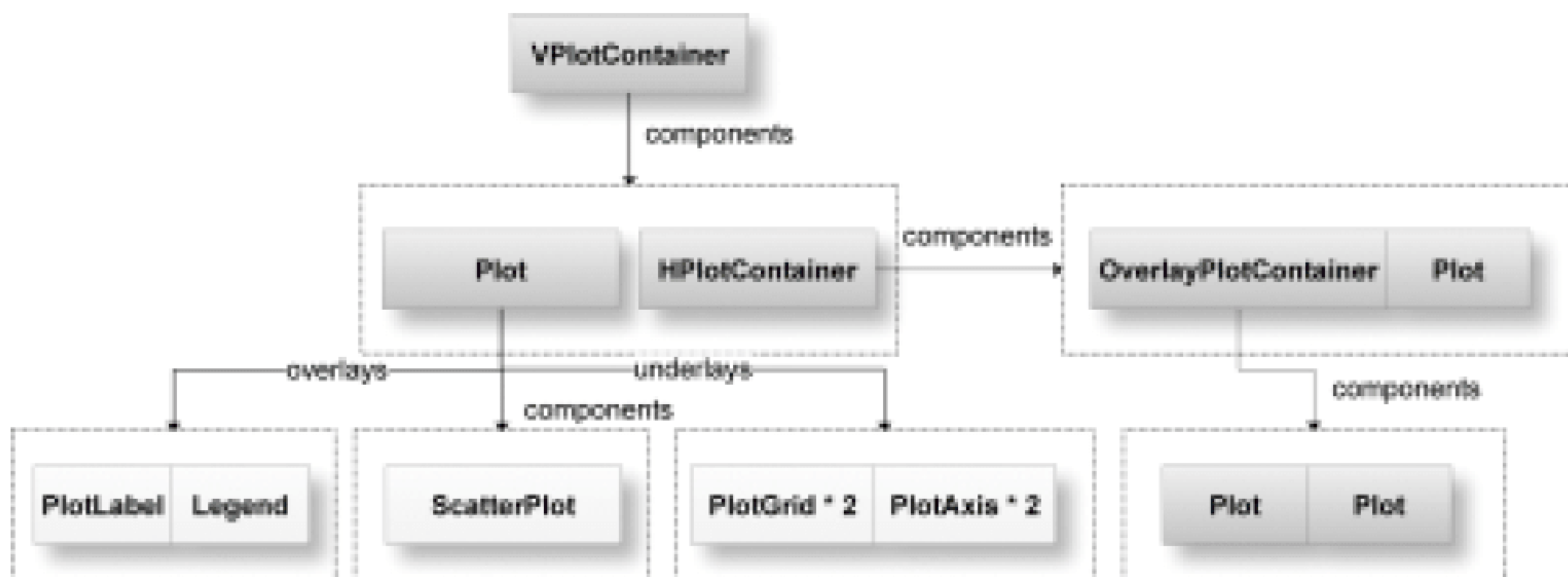


图 8-9 各个容器和绘图组件之间的关系

8.3 添加交互工具

Chaco 和 matplotlib 相比，最大的优点就是它提供了十分丰富的交互工具。在 IPython 中输入下面的语句，可以看到 Chaco 提供的所有工具，从工具的名称能大致猜出它们的用途：

```
>>> import enthought.chaco.tools.api as t
>>> [x for x in dir(t) if isinstance(getattr(t,x), type)]
['BetterSelectingZoom',
 'BetterZoom',
 'BroadcasterTool',
 'DataLabelTool',
 'DataPrinter',
 'DragTool',
 'DragZoom',
 [[省略]]
```

下面看看如何利用这些工具为图表添加各种各样的交互功能。

8.3.1 平移和缩放

平移和缩放是和图表进行交互的两个最基本的工具，下面的程序演示了如何在图表中添加它们：



chaco_tools_pan_zoom.py
添加平移和缩放功能



在 Chaco 3.4 中，DragZoom 工具存在一个 Bug，会导致通过鼠标拖动进行缩放无法正常工作。读者可以按照下面的说明进行修改。

用编辑器打开如下文件：

```
site-packages\Chaco-3.4.0-py2.6-win32.egg\enthought\chaco\tools\drag_zoom.py
```

在其中 DragZoom 类的 dragging()方法的最后添加如下代码：

```
self.zoom_in_x(zoom_x)
self.zoom_in_y(zoom_y)
self._prev_x = event.x      # <-- 添加此行代码
self._prev_y = event.y      # <-- 添加此行代码
return
```

程序的构造和前面介绍的例子基本相同，我们只针对增加的部分进行说明。首先需要载入这些交互工具：平移工具 PanTool、缩放工具 ZoomTool 和拖动缩放工具 DragZoom。

```
from enthought.chaco.tools.api import PanTool, ZoomTool, DragZoom
```

在初始化方法__init__()中，使用下面的程序为 Plot 对象添加交互工具：

```
plot.tools.append(PanTool(plot))
plot.tools.append(DragZoom(plot, drag_button="right",
    maintain_aspect_ratio=False))
plot.overlays.append(ZoomTool(plot))
```

每个工具都需要知道自己所处理的组件对象，这里在创建工具对象时，将 Plot 对象作为第一个参数传递给它。PanTool 和 DragZoom 对象被放入 Plot 对象的 tools 列表，而 ZoomTool 对象则被放入 overlays 列表。这是因为 PanTool 和 DragZoom 对象仅响应用户的键盘和鼠标事件，在界面上没有任何显示。而 ZoomTool 对象提供了多种缩放功能，用户可以使用它的“选择放大”功能，选择图表中的一块区域进行放大，因此它需要在图表上绘制表示选择区域的矩形，由于选区矩形在数据曲线之上显示，因此需要将它添加进 overlays 列表。

有了 PanTool 工具之后，可以在图表中按住鼠标左键拖动以改变 X-Y 轴的显示范围。有了 ZoomTool 工具，便可以使用鼠标滚轴对图表进行缩放。或者按一下“z”键，进入“选择放大”状态，此时光标变成一个放大镜，按住鼠标左键拖动可以选择要放大的区域，松开左键后，所选的区域将充满整个图表。一连串的“选择放大”操作将会被记录下来，按“Ctrl+Z”和“Ctrl+Y”可以在历史选区之间来回切换。而 DragZoom 工具则可以通过鼠标的拖动来实现 X 轴和 Y 轴的单独缩放。

可以通过设置工具对象的各种 Trait 属性，自定义它们的交互方式。例如程序中设置了 DragZoom 对象的 drag_button 和 maintain_aspect_ratio 属性。drag_button 属性指定拖动操作所使用的鼠标按键，为了和 PanTool 工具的按键相区别，这里设置为使用鼠标右键进行拖动缩放。maintain_aspect_ratio 属性为 False，表示 X 轴和 Y 轴不需要保持一定的比例，可以单独进行缩放。拖动缩放工具 DragZoom 的属性，如表 8-1 所示。

表 8-1　DragZoom 工具的属性

属　性　名	说　　明
drag_button	拖动用的鼠标按键："left"、"middle"、"right"
speed	拖动速度，缺省为 1.0
maintain_aspect_ratio	是否维持 X-Y 轴的比例，默认为 True
drag_pointer	拖动缩放时的鼠标光标，图标默认为"magnifier"
single_axis	是否只在一个轴上进行缩放
axis	缩放轴，single 为 True 时，axis 指定缩放的轴

平移工具 PanTool 的属性如表 8-2 所示。

表 8-2　PanTool 工具的属性

属　性　名	说　　明
drag_button	拖动用的鼠标按键："left"、"middle"、"right"
speed	拖动速度，默认为 1.0
constrain	是否锁定拖动方向，默认为 False
constrain_key	锁定拖动的修饰键：None、"shift"、"control"、"alt"
constrain_direction	锁定拖动方向：None、"x"、"y"
restrict_to_data	是否限制在数据范围之内，默认为 False

通过 drag_button 属性可以指定进入平移状态的鼠标按键，例如可以将拖动按键设置为中键。如果设置 constrain_key 属性为某个修饰键，按住此键进行平移时，将只能在 X 轴或 Y 轴方向上进行平移。也可以设置 constrain 属性为 True，并通过 constrain_direction 属性设置拖动方向，这样一来就只能在一个方向上进行平移。

缩放工具 ZoomTool 的部分属性如表 8-3 所示。

表 8-3　ZoomTool 工具的部分属性

属　性　名	说　　明
tool_mode	范围缩放的模式："box"、"range"
always_on	范围缩放模式是否自动启动，默认为 False
always_on_modifier	进入范围缩放模式的修饰键
enter_zoom_key	进入范围缩放模式的按键，默认为"z"
drag_button	选择缩放范围的鼠标按键："left"、"right"、None
axis	范围缩放的轴："index"、"value"。仅当 tool_mode 为"range"时有效
enable_wheel	是否开启鼠标滚轴缩放功能，默认为 True
wheel_zoom_step	鼠标滚轴缩放的缩放速度，默认为 1

缩放工具提供了两种缩放方式：鼠标滚轴缩放和范围缩放。而范围缩放又分为矩形模式"box"和轴范围模式"range"两种。用 enter_zoom_key 属性指定的按键进入范围模式，使用 drag_button 属性指定的鼠标按键选择范围。如果 always_on 为 Ture，用 always_on_modifier 属性指定的修饰键也可以进入范围模式。此外，还可以通过 pointer、color、alpha、border、border_size 等属性指定鼠标的光标以及绘制的范围矩形的颜色、透明度、边框及边框粗细。例如下面的程序为图表添加一个在 X 轴上进行放大的工具，按住 Ctrl 修饰键可进入 X 轴缩放模式：

```
plot.overlays.append(ZoomTool(plot, tool_mode="range", axis = "index",
    always_on=True, always_on_modifier="control"))
```

8.3.2 选取范围

Chaco 提供了许多对数据进行选择的工具。下面的程序使用范围选择工具 RangeSelection，制作了一个在 X 轴上进行动态缩放的交互式界面，效果如图 8-10 所示。用鼠标右键在上方图表中选择一个范围，下方图表中将显示所选择的部分，以便用户观察数据的细节部分。

chaco_tools_range_select.py
用范围选择工具观察数据的细节

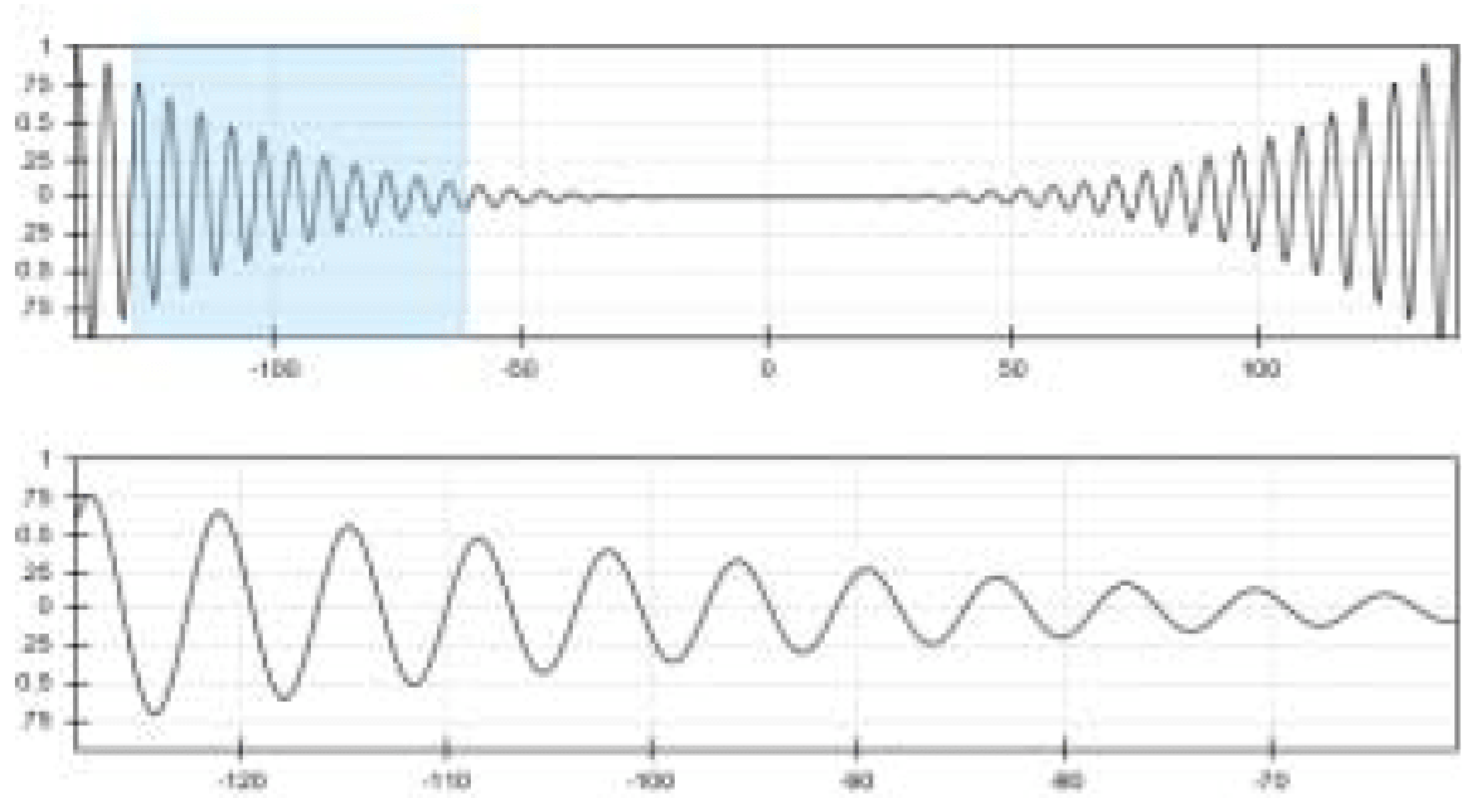


图 8-10 用范围选择工具观察数据的细节

```
from enthought.chaco.tools.api import RangeSelection, RangeSelectionOverlay
```

首先从工具库中载入 RangeSelection 和 RangeSelectionOverlay。RangeSelection 负责响应鼠标事件，进行范围选择；而 RangeSelectionOverlay 则负责将选区绘制到图表之上。

```
self.plot1 = plot1 = Plot(data, padding=25)
self.line1 = line1 = plot1.plot(("x", "y"), type="line")[0]  ❶
self.select_tool = select_tool = RangeSelection(line1) ❷
line1.tools.append( select_tool )
select_tool.on_trait_change(self._selection_changed, "selection") ❸
line1.overlays.append(RangeSelectionOverlay(component=line1)) ❹
self.plot2 = plot2 = Plot(data, padding=25)
plot2.plot(("x", "y"), type="line")[0]
self.plot = VPlotContainer(plot2, plot1)
```

上面是__init__()中进行图表配置的部分。我们创建了两个 Plot 对象——plot1 和 plot2，然后使用一个 VPlotContainer 对象将它们垂直排列。❶调用 plot1 的 plot()进行绘图，并用 line1 保存返回的 LinePlot 对象。❷为 line1 添加 RangeSelection 工具，由于 RangeSelection 工具只负责

事件处理，因此将它添加到 line1 的 tools 列表中。❸为选择工具 select_tool 的 selection 属性添加事件监听方法。当用户通过界面修改了选择范围时，选择工具的 selection 属性将发生变化，从而调用_selection_changed()方法。❹在 line1 的 overlays 列表中添加一个 RangeSelectionOverlay 对象，它将显示用户所选择的范围。注意，这里是将两个工具添加进 line1 对象而不是 plot1 对象，稍后将对此进行解释。

```
def _selection_changed(self):
    selection = self.select_tool.selection
    if selection != None:
        self.plot2.index_range.set_bounds(*selection)
    else:
        self.plot2.index_range.reset()
```

当选区发生变化时，_selection_changed()方法将被调用。它根据当前的选择范围，调用 plot2.index_range 的 set_bounds()或 reset()方法，修改下方图表中 X 轴的显示范围。

读者也许会有疑问：RangeSelectionOverlay 对象是如何知道 RangeSelection 对象的选区发生变化的？从程序中，我们找不到在二者之间建立联系的代码。它们之间的信息交互不是直接进行的，而是通过它们所共知的 line1 对象作为中间代理间接地进行交互。下面我们通过 IPython 分析一下它们是如何进行交互的。

首先，line1 是一个 LinePlot 对象，它有 index 和 value 两个属性，分别保存曲线的 X-Y 轴数据，它们都是 ArrayDataSource 对象：

```
>>> p.line1
<enthought.chaco.lineplot.LinePlot object at 0x0C1084E0>
>>> p.line1.index
<enthought.chaco.array_data_source.ArrayDataSource object at 0x0C108330>
```

ArrayDataSource 对象有一个 metadata 属性，用来存储对数据进行描述的数据，我们称之为元数据。metadata 属性是一个类似于字典的 TraitDictObject 对象：

```
>>> p.line1.index.metadata.keys()
['selection_masks', 'annotations', 'selections']
>>> p.line1.index.metadata["selections"]
(-127.95180722891567, -60.963855421686759)
>>> p.select_tool.selection
(-127.95180722891567, -60.963855421686759)
>>> p.line1.index.metadata["selection_masks"]
[array([False, False, False, False, False, False, False, False, False, ...])]
```

不同的工具会在 metadata 属性中添加不同的元数据，例如'selection_masks'和'selections'都是 RangeSelection 对象添加的元数据。其中，'selections'表示选区范围，而'selection_masks'是一个

布尔数组的列表，它表示数据中的每个数值是否在选区之内。

RangeSelection 对象的工作就是响应用户的操作，并对 metadata 属性中的元数据进行更新。而 RangeSelectionOverlay 对象则监视 metadata 属性中名为'selections'的元数据的变化，用它的最新值在图表上绘制选区。下面的程序获得它所监视的元数据名，由于默认值就是'selections'，因此无需对其进行配置。如果读者自己编写了工具类来修改别的元数据，可以通过 metadata_name 属性指定与它相关的元数据名。

```
>>> p.line1.overlays[0].metadata_name
'selections'
```

区域选择工具也有很多 Trait 属性，例如，如果将 RangeSelection 和 RangeSelectionOverlay 对象的 axis 属性都设置为'value'，那么它们将对 line1.value 的'selections'元数据进行处理，从而实现在 Y 轴上的选区选择。

由于 Plot 对象是一个容器，它本身没有 index 和 value 等属性，因此我们不能为 Plot 对象添加区域选择工具。整个系统的构造如图 8-11 所示。

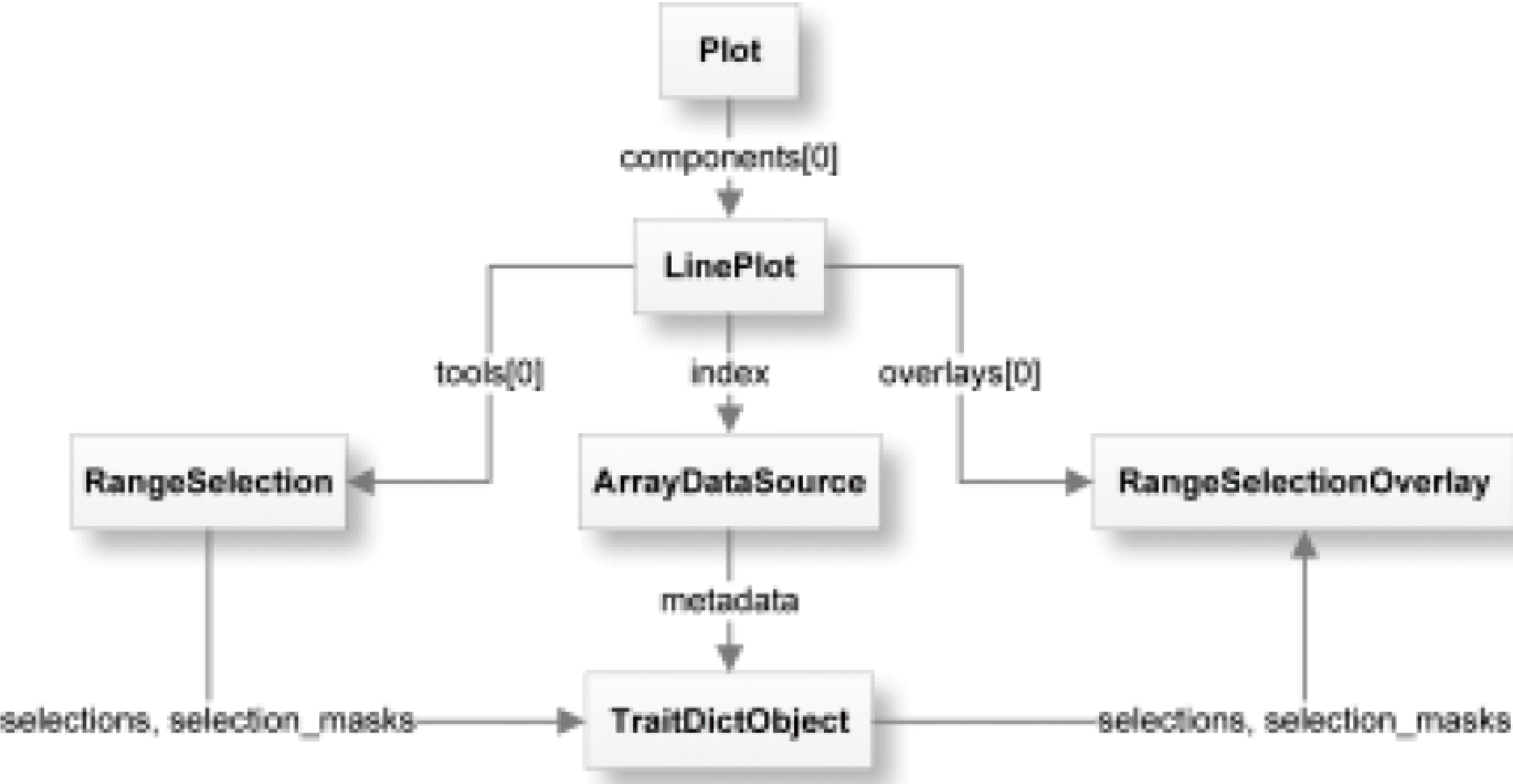


图 8-11　选择工具通过元数据传递信息

8.3.3　选取数据点

使用 ScatterInspector 和 ScatterInspectorOverlay 工具可以对图表中的单个数据点进行选取。图 8-12 是使用它们对数据进行选取的一个例子。我们可以使用绿色或红色两种颜色的选择工具对数据点进行选择。如果某个数据点同时被两个工具选中，那么它的颜色为红色和绿色的混合。两个工具所选择的数据点分别用两个列表控件显示。下面让我们分析一下这个程序。

chaco_tools_point_select.py
使用点选择工具选择数据点

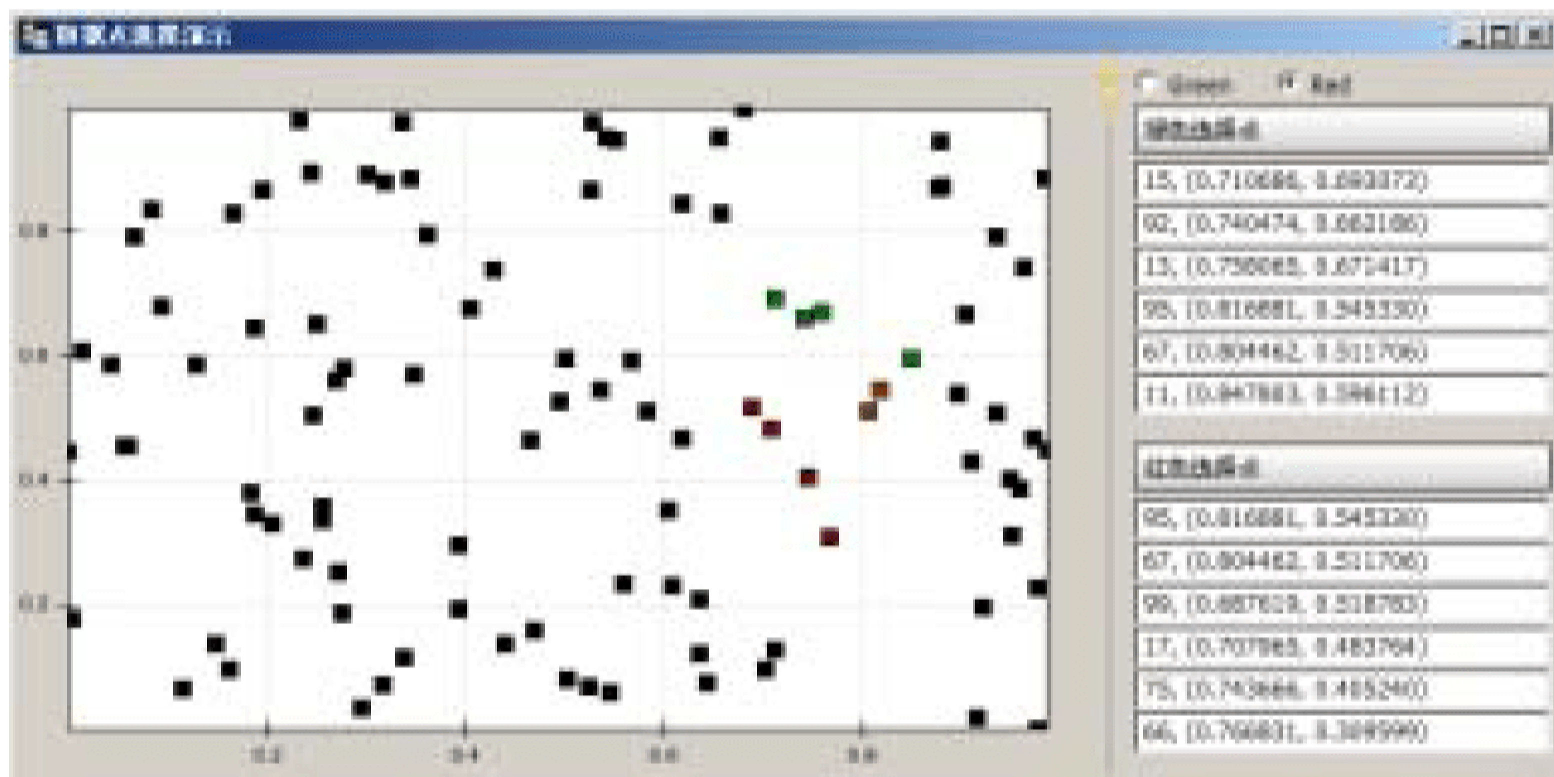


图 8-12　选取数据点，并标为红色和绿色

```
from enthought.chaco.api import ScatterInspectorOverlay
from enthought.chaco.tools.api import ScatterInspector
```

首先载入 ScatterInspector 和 ScatterInspectorOverlay，注意它们的载入位置不同。和前面介绍的 RangeSelection 和 RangeSelectionOverlay 一样，ScatterInspector 负责响应鼠标事件，将用户选择的点写入元数据中，ScatterInspectorOverlay 则根据元数据在 overlays 层绘制被选中的点的标识。

```
Colors = {
    "green":(0, 1, 0, 0.5),
    "red":(1, 0, 0, 0.5)
}
```

接下来定义了一个颜色字典，以颜色名为键，而值是颜色值。将颜色的透明度设置为 0.5，这样一来，当两个颜色叠加时便能够混合出别的颜色。

```
class PointSelectionDemo(HasTraits):
    color = Enum(Colors.keys())
    green_selection = List()
    red_selection = List()
    plot = Instance(Plot)
    data = Instance(ArrayPlotData)
```

PointSelectionDemo 类有 4 个 Trait 属性。其中，color 属性用于设置选择工具，green_selection 和 red_selection 属性用于显示被两个选择工具选中的数据。接下来跳过视图定义部分，直接查看__init__()中设置图表的部分：

```
plot = Plot(data, padding=25)
self.scatter = scatter = plot.plot(("x", "y"), type="scatter", marker_size=4)[0] ❶
```

```
    self.select_tools = {}
    for i, c in enumerate(Colors.keys()):
        hover_name = "hover_%s" % c
        selection_name = "selections_%s" % c
        self.select_tools[c] = ScatterInspector(scatter,  ❷
            hover_metadata_name=hover_name,
            selection_metadata_name=selection_name)
        scatter.overlays.append(ScatterInspectorOverlay(scatter,  ❸
            hover_metadata_name = hover_name,
            selection_metadata_name=selection_name,
            hover_color = "transparent",
            hover_outline_color = c,
            hover_marker_size = 6,
            hover_line_width = 1,
            selection_color = Colors[c],
        ))
    scatter.active_tool = self.select_tools[self.color] ❹
    scatter.index.on_trait_change(self.selection_changed, 'metadata_changed') ❺
    self.plot = plot
    self.data = data
```

❶由于 plot()的 type 参数为"scatter"，因此它返回一个 ScatterLine 对象的列表。❷在循环体中创建 ScatterInspector 对象，并设置它的 hover_metadata_name 和 selection_metadata_name 属性，这两个属性决定它所使用的元数据名称。当鼠标移动到某个数据点之上时，将在元数据字典中添加一个键为 hover_metadata_name 的元素；而选择的数据将存放在元数据字典中以 selection_metadata_name 为键的列表中。这里使用循环创建了两个 ScatterInspector 对象，它们对应的元数据名分别为"hover_green"、"hover_red"、"selections_green"和"selections_red"。所创建的 ScatterInspector 对象并不直接存放到 scatter 对象的 tools 列表中，而是先用一个字典 select_tools 将它们保存起来。

❸创建 ScatterInspectorOverlay 对象，将它的两个用来定义元数据名的属性设置成和 ScatterInspector 对象一样。ScatterInspectorOverlay 对象有许多 Trait 属性用来设置其绘图效果。以“hover_”开头的属性用于设置名为 hover_metadata_name 的元数据的显示效果；而以“selection_”开头的属性则用于设置名为 selection_metadata_name 的元数据的显示效果。

下面在 IPython 中查看由这些工具产生的元数据。在 IPython 中运行程序之后，分别用红色和绿色选择工具在图表中选择几个点，并让鼠标停留在某个数据点之上。按 Alt+Tab 切换回 IPython，并运行下面的语句：

```
>>> p.scatter.index.metadata
{'annotations': [],
 'hover_red': [73],
 'selections': [],
```

```
    'selections_green': [15, 92, 13, 95, 67, 11],
    'selections_red': [95, 67, 99, 17, 75, 66]}
```

这表示红色选择工具的鼠标正在下标为 73 的数据点之上，并且选择了下标为 95、67、99、17、75、66 的数据点。绿色选择工具则选择了下标为 15、92、13、95、67、11 的数据点。其中，下标为 95 和 67 的数据点同时被两个工具选中。

❹我们不能将两个 ScatterInspector 对象都添加进 scatter 的 tools 列表中，因为它们会同时响应鼠标事件。只能将被选中的那个选择工具设置给 scatter 的 active_tool 属性。通过修改 active_tool 属性，可以使用不同的选择工具响应鼠标事件。因此，当通过界面上的单选按钮修改 PointSelectionDemo 对象的 color 属性时，我们需要修改 ScatterPlot 对象的 active_tool 属性：

```
def _color_changed(self):
    self.scatter.active_tool = self.select_tools[self.color]
```

❺当 scatter.index 的元数据发生变化时，将会触发其 metadata_changed 事件，通过添加处理此事件的方法，可以在用户的程序中监视元数据的变化。由于 scatter.value 和 scatter.index 的元数据会同时发生变化，因此只需要处理其中的一个事件即可。当鼠标移入、移出或选中某个数据点时，元数据都会发生变化，从而调用 selection_changed()方法。

```
def selection_changed(self):
    x = self.scatter.index.get_data() ❶
    y = self.scatter.value.get_data() ❷
    metadata = self.scatter.index.metadata
    selection = metadata.get("selections_green", []) ❸
    self.green_selection = ["%d, (%f, %f)" % (s, x[s], y[s]) for s in selection]
    selection = metadata.get("selections_red", []) ❹
    self.red_selection = ["%d, (%f, %f)" % (s, x[s], y[s]) for s in selection]
```

在 selection_changed()中，❶和❷分别获得表示数据点的 X-Y 轴坐标值的数组，❸和❹分别使用名为"selections_green"和"selections_red"的元数据从数据数组中获得被选中的数据点的坐标。

8.3.4 套索工具

使用套索工具可以用鼠标绘制一个任意形状的选区，并选取落入此区域中的数据点。下面的程序演示如何使用套索工具选择数据，效果如图 8-13 所示。



chaco_tools_lasso_selection.py
用套索工具选择数据点

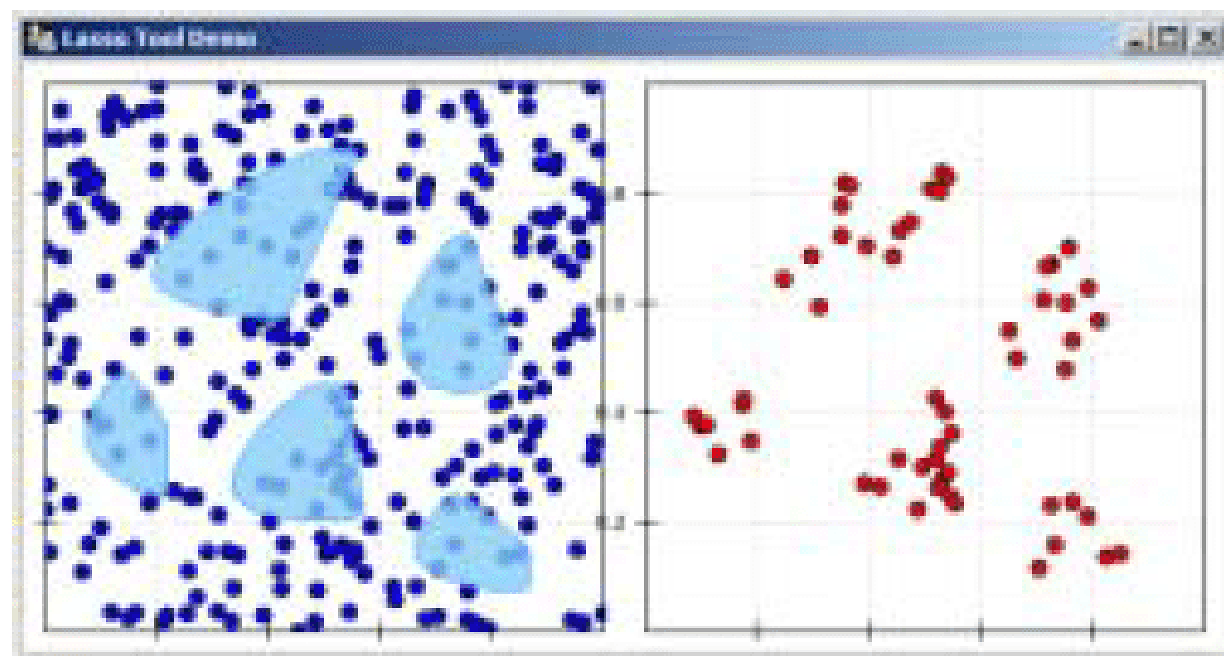


图 8-13　用套索工具选择数据点

```
from enthought.chaco.api import LassoOverlay
from enthought.chaco.tools.api import LassoSelection
```


套索工具也分为两部分：LassoSelection 对象对用户的鼠标操作进行处理，LassoOverlay 对象则显示用户所绘制的套索区域。

在__init__()中，首先创建数据对象 data，并为其添加名为"x"、"y"、"x2"、"y2"的 4 个数组。我们省略这段程序，直接查看对 Plot 对象进行设置的部分：

```
plot1 = Plot(data, padding=10)
scatter_plot1 = plot1.plot(("x", "y"), type="scatter", marker="circle",
    color="blue")[0]
self.lasso = LassoSelection(scatter_plot1, incremental_select=True,  ❶
    selection_datasource=scatter_plot1.index)
self.lasso.on_trait_change(self._selection_changed, 'selection_changed') ❷
scatter_plot1.tools.append(self.lasso)
scatter_plot1.overlays.append(LassoOverlay(scatter_plot1,
    lasso_selection=self.lasso)) ❸
plot2 = Plot(data, padding=10)
plot2.index_range = plot1.index_range
plot2.value_range = plot1.value_range
plot2.plot(("x2", "y2"), type="scatter", marker="circle", color="red")
```

❶创建处理鼠标操作的 LassoSelection 对象。incremental_select 参数为 True，表示当套索区域改变时，将会立即修改其 selection 属性。如果为 False，只有在松开鼠标按键完成选区时才更新 selection 属性。LassoSelection 对象和前面介绍的工具有所不同，需要通过 selection_datasource 属性指定与之相关的数据对象。此数据对象中名为"selection"的元数据是一个遮罩数组，其中值为 1 的元素表示数据对象中对应下标的数据点被套索选中，为 0 表示没有被选中。❷为 LassoSelection 对象的 selection_changed 事件设置事件处理方法，当 selection 属性改变时，selection_changed 事件将会被触发。

❸创建显示套索区域的 LassoOverlay 对象。由于 LassoSelection 对象不在数据对象的元数据中存储表示套索区域的多边形数据，因此需要设置 LassoOverlay 对象的 lasso_selection 属性，指定与之对应的 LassoSelection 对象。

```
def _selection_changed(self):
    index = np.array(self.lasso.selection_datasource.metadata["selection"],
            dtype=np.bool)
    self.data["x2"] = self.data["x"][index]
    self.data["y2"] = self.data["y"][index]
```

最后在套索工具的事件处理方法中，将元数据中的遮罩数组转换为布尔数组，并使用它分别对名为"x"和"y"的数组进行下标选择，更新名为"x2"、"y2"的数组。由于右侧的 Plot 对象和名为"x2"和"y2"的数据相关联，因此当它们改变时，图 8-13 中右侧的图会自动更新。整个系统的构造如图 8-14 所示(见文前彩插)。

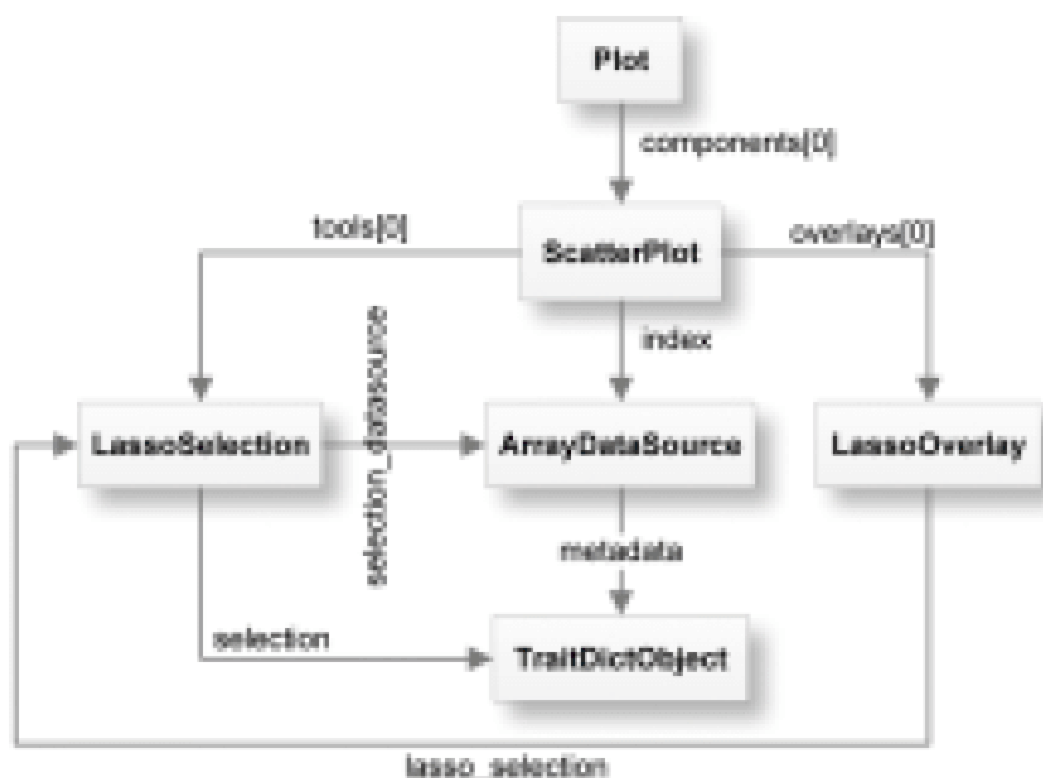


图 8-14　使用套索工具时的系统构造

8.4　二 次 开 发

为了开发自己的 Chaco 工具，需要了解 Chaco 的内部结构。Chaco 内部使用了 Enthought 公司发布的 Enable、Kiva、Traits、TraitsUI 等几个扩展库，它们之间的依存关系如图 8-15 所示。

Chaco 是制作二维交互式图表的绘图库，其中定义了各种类型的图表类、表示图表数据的数据源类，以及表示数据轴、网格的图表工具类。

Chaco 的所有绘图类都建立在 Enable 库的 Component 类基础之上。Component 类向 Chaco 提供了一个统一的管理绘图对象的平台，而 Enable 库中的 ComponentEditor 则为后台 GUI 库提供了一个可以进行绘图的控件。它使我们可以将图表嵌入到由 wxPython 或 PyQt4 制作的界面程序中。

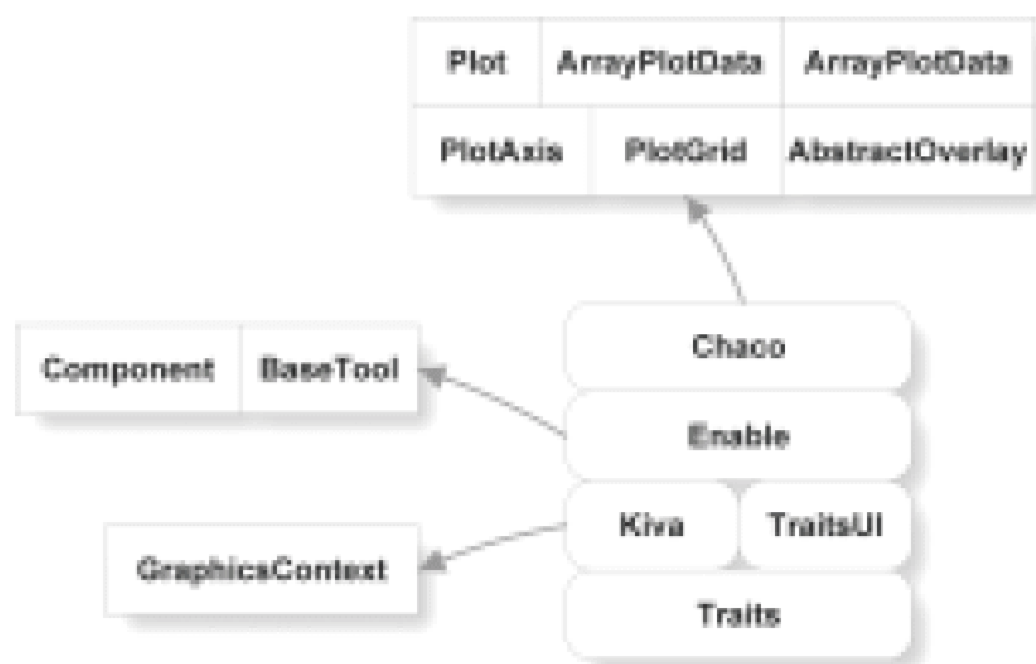


图 8-15　Chaco 使用的库

Kiva 提供了一个统一的绘图 API 接口，它可以使用多种图形绘制库绘制图形。对于每个图形库，Kiva 都定义了一个 GraphicsContext 类以封装所有的绘图细节。Enable 通过 GraphicsContext 在 Component 上绘制图形。

在 Chaco 和 Enable 中，大多数类都从 HasTraits 继承，它们依靠 Traits 和 TraitsUI 库提供的强大功能，实现各种复杂的逻辑处理。

8.4.1　用 Kiva 库在数组上绘图

Kiva 库可以单独用来绘图，还可以在多种后台上进行绘图，例如 wxPython、PyQt4、PDF、SVG 等等。在 Python(x,y)的安装目录下能够找到一些使用 Kiva 绘图的例子：

```
c:\pythonxy\doc\Enthought Tool Suite\Enable\kiva
```

Kiva 还能直接在 NumPy 数组上进行绘图，下面的例子演示了这个功能，效果如图 8-16 所示(见文前彩插)。

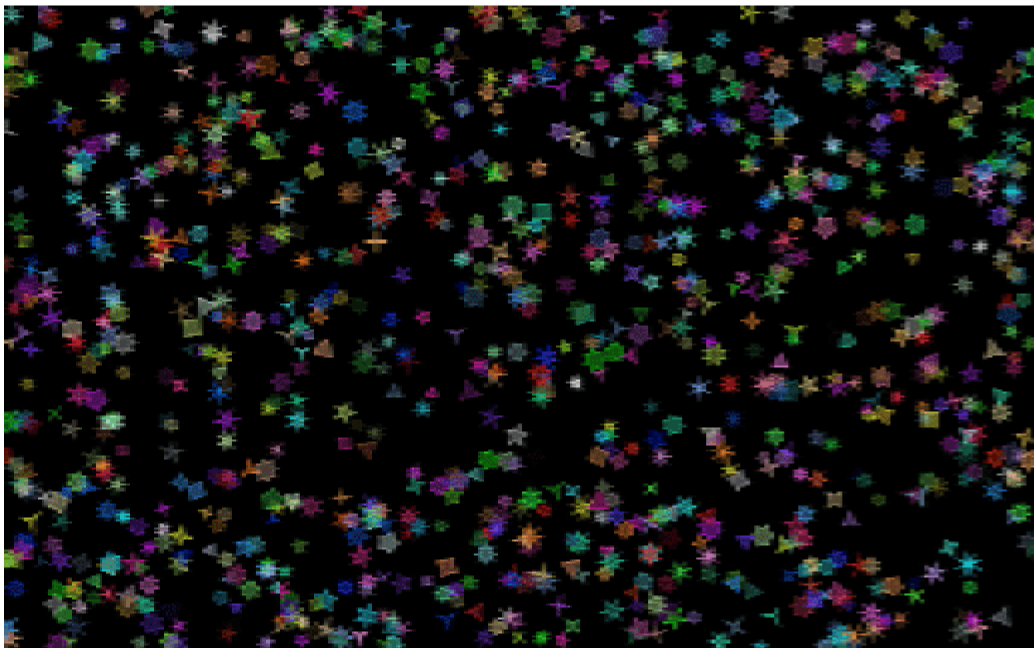


图 8-16　用 Kiva 在数组上绘制星星图案

chaco_kiva_stars.py
用 Kiva 在数组上绘制星星图案

```
import numpy as np
from enthought.kiva.image import GraphicsContext
```

为了在数组上进行绘图，需要从 Kiva 库的 image 模块载入 GraphicsContext 类，它提供在内存中的图像(数组)上绘制图案的各种方法。

```
img = np.zeros((500,800,4), dtype=np.uint8) ❶
img[:,:,:3] = 0 # 全黑图像
img[:,:,3] = 255
gc = GraphicsContext(img) ❷
for i in range(1000):
    randint = np.random.randint
    rand = np.random.rand
    draw_star(gc, randint(800), randint(500), randint(5,10),  ❸
        (rand(),rand(),rand()), rand()*2*np.pi, randint(3,9), rand()*0.6+0.1)
gc.save("stars.png")  ❹
```



目前，GraphicsContext 类只支持在 3 通道或 4 通道的图像数组上进行绘图。使用单通道的图像数组虽然可以创建 GraphicsContext 对象，但是却无法在它上面进行绘图。

❶创建一个形状为(500, 800, 4)的无符号单字节整数数组 img。如果把此数组当做一幅图像，那么第 0 轴的长度 500 就是图像的高度，第 1 轴的长度 800 则是图像的宽度，第 2 轴的长度 4 表示图像的颜色有 4 个通道——红、绿、蓝和 alpha 通道。使用数组 img 创建一个 GraphicsContext 对象 gc，通过调用 gc 的绘图方法在数组上进行绘图。在接下来的循环中，调用 draw_star()在 gc 上绘制各种颜色的星星图案。❹最后调用 gc 的 save()方法将数组保存成 PNG 图像文件。下面是绘制星星的程序：

```
def star_polygon(x, y, r, theta, n, s): ❺
    angles = np.arange(0, 2*np.pi+2*np.pi/2/n, 2*np.pi/2/n) + theta
    xs = r * np.cos(angles)
    ys = r * np.sin(angles)
    xs[1::2] *= s
    ys[1::2] *= s
    xs += x
    ys += y
    return np.vstack([xs, ys]).T

def draw_star(gc, x, y, r, c, theta, n, s): ❻
    gc.save_state()
    gc.set_antialias(True)
    gc.set_stroke_color(c + (0.8,))
    gc.set_fill_color(c + (0.4,))
```

```
        gc.lines(star_polygon(x, y, r, theta, n, s))
        gc.draw_path()
        gc.restore_state()
```

❺star_polygon()通过一系列的参数，计算出表示星星图案的多边形的顶点坐标。其中：x 和 y 是多边形的中心坐标；r 是中心到多边形顶点的距离，也就是星星的大半径；theta 是多边形顶点的起始弧度；n 是星星顶角的个数；s 是星星小半径和大半径的比例。它返回的是一个描述封闭多边形顶点坐标的形状为(N+1, 2)的二维数组，其中 N 是多边形的顶点数。例如，五角星有 10 条边，因此数组的形状为(11,2)，其中首尾两个点的坐标是完全一样的。

❻draw_star()在 gc 上绘制星星，它调用了 gc 的一系列方法来完成绘制工作。绘图的一般步骤如下：

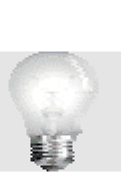
调用 save_state()保存当前的绘图状态，绘图状态包括当前的颜色、线型、粗细等许多与绘图相关的配置。

调用一系列 set_*()方法设置各种绘图属性，例如 set_antialias()配置反锯齿效果、set_stroke_color()配置线条颜色、set_fill_color()配置填充颜色。

调用各种绘图方法创建图案的路径，例如：lines()创建多条连续的直线，move_to()、line_to()创建单条直线，arc()创建圆弧等。调用这些方法时并没有真正绘图，而是在 gc 内部的路径对象中添加数据。

调用 draw_path()进行绘图，它使用 gc 的路径对象和当前的绘图属性在数组上绘图。绘图完成之后它会自动清空路径对象，以便创建下一个路径。draw_path()会对路径进行填充并进行描边，而 fill_path()只进行填充，stroke_path()只进行描边。通过 gc._get_path()可以获得路径对象，而通过路径对象的_vertices()可以获得路径的顶点信息。

最后调用 restore_state()，恢复到调用 save_state()之前的绘图状态。

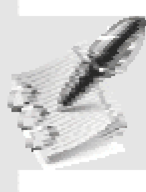


Kiva 库缺乏文档，如果读者想深入了解各种绘图函数的用法，可以在 Chaco 的目录下对所有的 Python 文件搜索“gc.”。这样能搜索到 Chaco 库的程序中使用 GraphicsContext 对象绘图的程序。

8.4.2 Enable 库的组件

Chaco 的所有绘图类都建立在 Enable 库的 Component(组件)类的基础之上。我们也可以直接使用 Component 制作交互式绘图程序。下面是一个实例，图 8-17 是它的界面截图。界面的用法如下：

- 通过窗口上方的工具栏可以修改星星的顶角数和颜色。
- 按住鼠标左键并拖动可绘制星星。
- 将鼠标移动到某个星星上之后，使用鼠标滚轴可以调节星星的大小半径的比例。
- 按住右键拖动可改变星星的位置。



chaco_enable_stars.py
用 Enable 库的组件制作绘制星星的界面程序



图 8-17　用 Enable 库的组件制作的绘制星星的程序界面

由于程序较长，我们只详细介绍其中和 Component 类相关的部分。

```
class Star(HasTraits):
    x = Float
    y = Float
    r = Float
    theta = Float
    n = Range(3, 10)
    s = Float
    c = Tuple
    def polygon(self):
        return star_polygon(self.x, self.y, self.r, self.theta, self.n, self.s)
```

首先，创建一个 Star 类保存和星星相关的所有属性，这些属性就是前面介绍的 star_polygon() 的参数。通过 Star 对象的 polygon()方法可以得到表示星星形状的多边形。

```
class StarComponent(Component):
    stars = List(Star)
    star_color = Color((255,255,255))
    edges = Range(3, 10, 5)
    sx = Float # 移动开始时的星星中心 X 坐标
    sy = Float # 移动开始时的星星中心 Y 坐标
    mx = Float # 移动开始时的鼠标 X 坐标
    my = Float # 移动开始时的鼠标 Y 坐标
    moving_star = Instance(Star)
```

```
    event_state = Enum("normal", "drawing", "moving")
```

StarComponent 类是实现星星绘制及响应用户操作的组件类，它从 Component 继承。它的各个 Trait 属性的含义如表 8-4 所示。

表 8-4　StarComponent 类的 Trait 属性

属　性　名	说　　明
stars	保存所有星星的列表
star_color	当前星星的颜色
edges	当前星星的顶角数
sx, sy	开始移动时的星星中心坐标
mx, my	开始移动时的鼠标坐标
moving_star	正在移动的星星对象
event_state	当前的事件状态

Component 对象具有绘图以及响应键盘和鼠标事件的能力。为了方便编写各种事件处理方法，Component 的父类 Interactor 提供了一种名为“事件状态”的机制。每个 Component 对象都有一个 event_state 属性，我们称之为“事件状态”。当界面中与组件对应的区域发生某个事件时，它将根据当前的事件状态和事件名，调用组件类中相应的事件处理方法。例如，如果当前的事件状态是"normal"，而事件名为"left_down"(鼠标左键按下)，那么如果在组件类中定义了名为“normal_left_down”的方法，该方法将被调用。下面是从 Enable 库的“interactor.py”文件中找到的事件名，它们的含义可以很容易根据名称猜测出来，因此这里就不多做解释了：

```
left_down, left_up, left_dclick,
right_down, right_up, right_dclick,
middle_down, middle_up, middle_dclick,
mouse_move, mouse_wheel,
mouse_enter, mouse_leave,
key_pressed,
dropped_on, drag_over, drag_enter, drag_leave
```

下面列出 StarComponent 类中定义的所有事件处理方法。在这些方法中，会修改 event_state 属性，实现不同状态下对不同事件的响应，事件状态的漂移如图 8-18 所示。

```
def normal_left_down(self, event):
    "添加一个 Star 对象到 stars 列表中，并切换到 drawing 状态"
    self.stars.append(
        Star(x=event.x, y=event.y, r=0, theta=0, n=self.edges,
            s = 0.5, c=convert_color(self.star_color)))
    self.event_state = "drawing"
    self.request_redraw()
```

```
def drawing_mouse_move(self, event):
    "修改 stars 中最后一个 Star 对象的半径和起始角度"
    star = self.stars[-1]
    star.r = np.sqrt((event.x-star.x)**2+(event.y-star.y)**2)
    star.theta = np.arctan2(event.y-star.y, event.x-star.x)
    self.request_redraw()
def drawing_left_up(self, event):
    "完成一个星形的绘制，回到 normal 状态"
    self.event_state = "normal"
def normal_mouse_wheel(self, event):
    "找到包含鼠标坐标的星形，并修改其半径比例"
    star = self.find_star(event.x, event.y)
    if star is not None:
        star.s += event.mouse_wheel * 0.02
        if star.s < 0.05: star.s = 0.05
        self.request_redraw()
def normal_right_down(self, event):
    "找到包含鼠标坐标的星形，用 moving_star 属性保存它，并进入 moving 状态"
    star = self.find_star(event.x, event.y)
    if star is not None:
        self.mx, self.my = event.x, event.y # 记录鼠标位置
        self.sx, self.sy = star.x, star.y # 记录星形的中心位置
        self.moving_star = star
        self.event_state = "moving"
def moving_mouse_move(self, event):
    "修改 moving_star 的 x、y 坐标，实现星形的移动"
    self.moving_star.x = self.sx + event.x - self.mx
    self.moving_star.y = self.sy + event.y - self.my
    self.request_redraw()
def moving_right_up(self, event):
    "移动操作结束，回到 normal 状态"
    self.event_state = "normal"
```

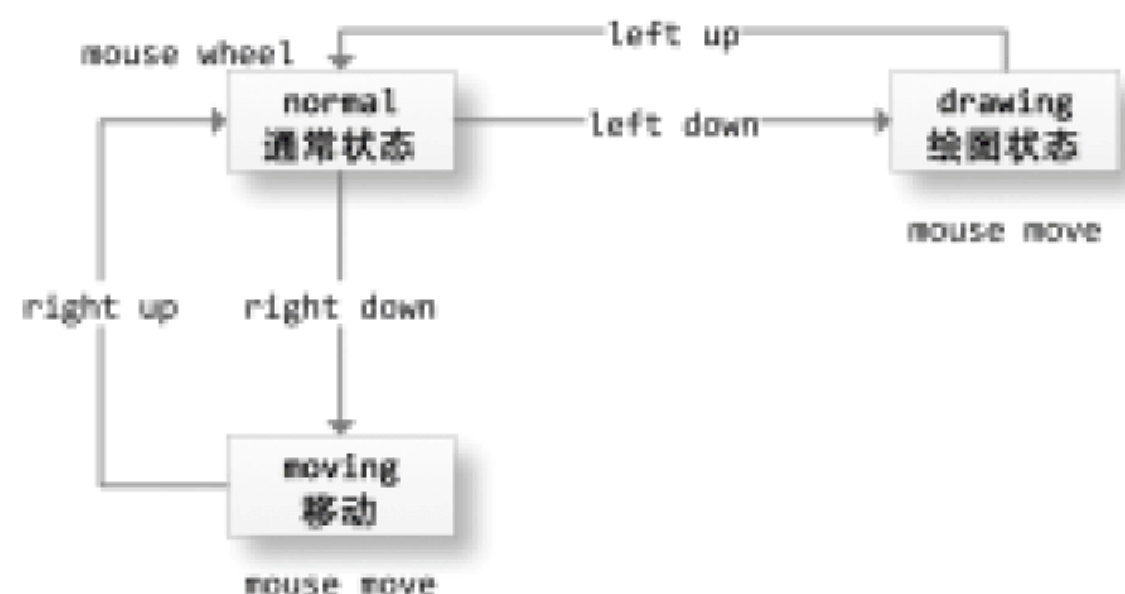


图 8-18　StarComponent 类的事件状态漂移

在事件处理方法中，通过修改 event_state 属性的值，实现了事件状态的漂移。StarComponent 对象共有三种事件状态："normal"、"drawing"和"moving"。在“normal”状态下按下鼠标左键会调用 normal_left_down()，此方法会修改 event_state 属性，进入“drawing”状态。而按下鼠标右键则会调用 normal_right_down()，并进入"moving"状态。

每个事件处理方法都有一个 event 参数，它描述和事件有关的详细信息。在 Enable 库的“events.py”文件中可以找到这些事件对象的定义。例如 MouseEvent 是鼠标事件对象，它的 x、y 属性是事件发生时的鼠标坐标，它还有 alt_down、control_down 等属性，表示辅助键是否被按下。

当保存星形对象的 stars 列表发生变化，或者其中的元素发生变化时，都需要调用组件的 request_redraw()方法刷新绘图。而具体的绘图工作在_draw_overlay()中进行，此方法会在需要重新绘图时自动被调用。由方法名可知，它是在"overlay"绘图层绘制图形。

```
def _draw_overlay(self, gc, view_bounds=None, mode="normal"):
    gc.clear((0,0,0,1)) #填充为全黑
    gc.save_state()
    for star in self.stars:
        draw_star(gc, star.x, star.y, star.r, star.c, star.theta, star.n, star.s)
        gc.draw_path()
    gc.restore_state()
```

为了找到使用鼠标选中的星形，我们需要一个判断点是否在多边形之内的函数。幸好 Kiva 库中已经有实现此功能的函数 points_in_polygon()，它的第一个参数可以是一个点，也可以是一系列的点；第二个参数是一个表示多边形的顶点数组，它返回元素值为 0 或 1 的数组，1 表示对应的点在多边形内部。使用此函数，我们很容易编写出找到包含坐标点(x,y)的星星的 find_star()方法：

```
def find_star(self, x, y):
    from enthought.kiva.agg import points_in_polygon
    for star in self.stars[::-1]:
        if points_in_polygon((x, y), star.polygon()):
            return star
    return None
```

最后看一下表示主界面的 StarDesign 类。在其视图定义中，我们直接使用 box 对象的 edges 和 star_color 属性在界面中创建编辑器：

```
class StarDesign(HasTraits):
    box = Instance(StarComponent)

    view = View(
        HGroup(Item("object.box.edges", label=u"顶角数"),
               Item("object.box.star_color", label=u"颜色")),
```

```
        Item("box", editor=ComponentEditor(),show_label=False),
        resizable=True,
        width = 600,
        height = 400,
        title = u"星空设计"
    )

    def __init__(self, **traits):
        super(StarDesign, self).__init__(**traits)
        self.box = StarComponent()
```

8.4.3 设计圆形选择工具

Chaco 没有使用圆形进行选择的工具，作为二次开发的实例，本节将介绍如何自己动手编写一个圆形选择工具。仿照前面介绍的选择工具，我们将编写两个类——CircleSelection 和 CircleSelectionOverlay，它们分别实现鼠标事件的处理和圆形的绘制。它们通过绘图对象的 index 属性的名为'circle_center'和'circle_radius'的元数据进行数据交互。效果如图 8-19 所示。



chaco_tools_circle.py
自己设计圆形选择工具

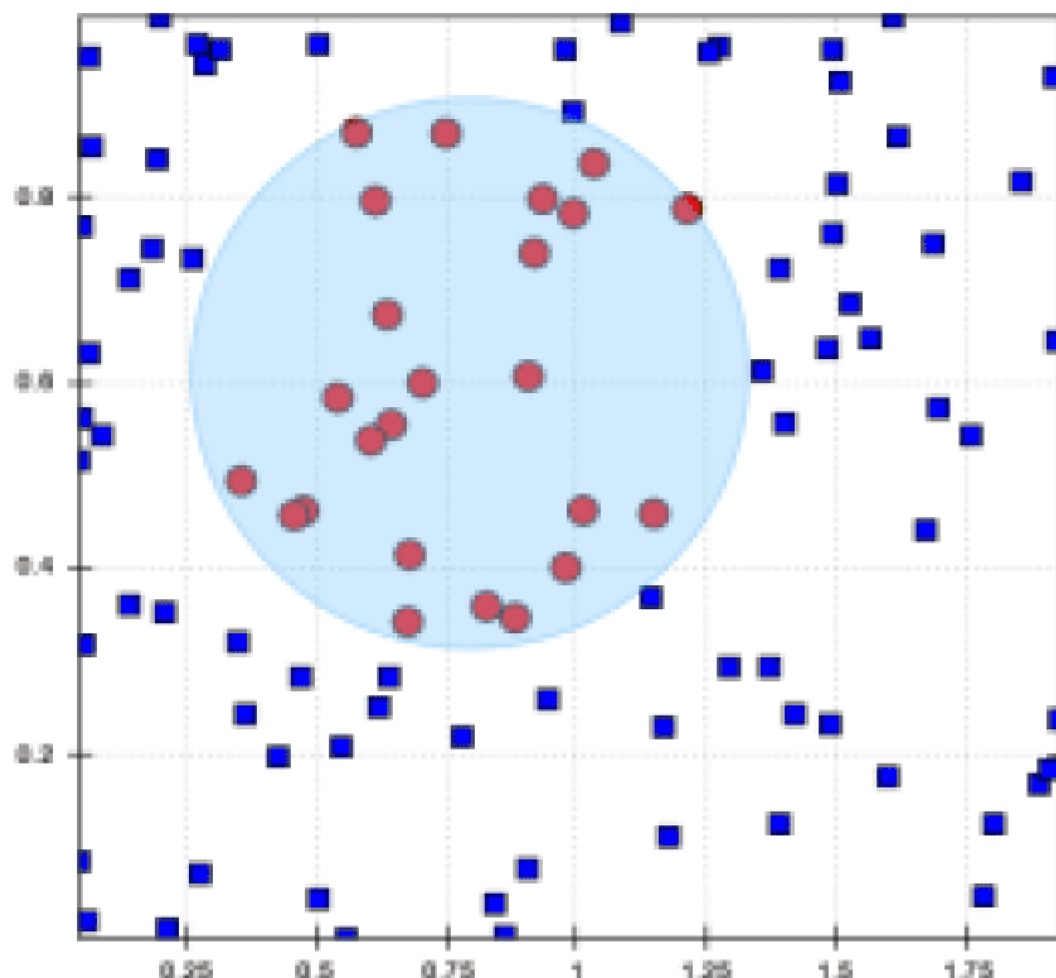


图 8-19　圆形选择工具

下面详细分析源程序。首先是用于绘制圆形的 CircleSelectionOverlay 类：

```
class CircleSelectionOverlay(AbstractOverlay): ❶
    metadata = Property(depends_on = 'component') ❷
```

```
    def _get_metadata(self):
        return self.component.index.metadata

    def overlay(self, component, gc, view_bounds=None, mode="normal"):
        if self.metadata.has_key('circle_center'):
            x, y = self.metadata['circle_center'] ❸
            r = self.metadata['circle_radius']
            gc.save_state()
            gc.set_alpha(0.4)
            gc.set_fill_color(color_table["lightskyblue"])
            gc.set_stroke_color(color_table["dodgerblue"])
            gc.set_line_width(1)
            gc.set_line_dash(None)
            gc.arc(x, y, r, 0.0, 2*np.pi) ❹
            gc.draw_path()
            gc.restore_state()
```


❶CircleSelectionOverlay 对象将放到绘图对象的 overlays 列表中，因此它从 AbstractOverlay 继承。AbstractOverlay 中定义了一个 overlay()方法，我们只需覆盖此方法，在其中完成圆形的绘制即可。

❷为了使程序简洁，定义了一个名为 metadata 的 Property 属性，它得到的是 component 属性中 index 数据对象的元数据。component 属性在父类 AbstractOverlay 中定义，它是创建 CircleSelectionOverlay 对象时所传递的第一个对象，通常就是拥有此 CircleSelectionOverlay 对象的绘图对象。

❸读取名为'circle_center'和'circle_radius'的元数据，它们分别表示圆形的中心和半径。由于这两个元数据保存的是屏幕坐标系中的坐标和长度，因此不需要进行任何坐标变换，可以直接使用它们绘制圆形。

❹圆形路径通过 gc 对象的 arc()方法创建，arc()实际创建的是圆弧路径，这里将圆弧的起始角度和终止角度设置为 0 和 2π，这样就绘制出一个完整的圆形。

CircleSelectionOverlay 类很简单，而响应用户操作的工作都在 CircleSelection 类中完成。CircleSelection 对象负责根据用户的操作更新'circle_center'和'circle_radius'元数据：

```
class CircleSelection(AbstractController): ❶
    metadata = Property(depends_on = 'component')
    event_state = Enum('normal', 'selecting', 'selected', 'moving') ❷
    selection_update = Event ❸
    x = Float # 圆心的 X 坐标
    y = Float # 圆心的 Y 坐标
    r = Float # 半径
    mx = Float # 移动开始时鼠标的 X 坐标
    my = Float # 移动开始时鼠标的 Y 坐标
```

```
    x0 = Float # 移动开始时圆心的 X 坐标
    y0 = Float # 移动开始时圆心的 Y 坐标

    def _get_metadata(self):
        return self.component.index.metadata
```

❶CircleSelection 类从 AbstractController 继承，在 AbstractController 中定义了 component 属性，在创建 CircleSelection 对象时，我们将绘图对象传递给它。

❷event_state 属性是一个 Enum 类型，允许有 4 种事件状态，分别是：

- 'normal'：通常状态，此时没有圆形选区。
- 'selecting'：正在进行选择，此时将根据鼠标的位置设置圆形选区的半径。
- 'selected'：完成圆形选区的选择操作。
- 'moving'：正在移动圆形选区。

❸selection_update 是一个 Event 属性，用来通知其他对象选区发生了变化。

```
def normal_left_down(self, event):
    self.x, self.y  = event.x, event.y
    self.metadata['circle_center'] = self.x, self.y
    self.metadata['circle_radius'] = 0
    self.event_state = 'selecting'
def selecting_mouse_move(self, event):
    self.r = np.sqrt((self.x-event.x)**2 + (self.y-event.y)**2)
    self.metadata['circle_radius'] = self.r
    self._update_selection()
def selecting_left_up(self, event):
    self.event_state = 'selected'
def selected_left_down(self, event):
    r = np.sqrt((self.x-event.x)**2 + (self.y-event.y)**2)
    if r > self.r:
        del self.metadata['circle_center']
        del self.metadata['circle_radius']
        del self.metadata['selections']
        self.selection_update = True
        self.event_state = 'normal'
    else:
        self.mx, self.my = event.x, event.y
        self.x0, self.y0 = self.x, self.y
        self.event_state = 'moving'
def moving_mouse_move(self, event):
    self.x = self.x0 + event.x - self.mx
    self.y = self.y0 + event.y - self.my
    self.metadata['circle_center'] = self.x, self.y
    self._update_selection()
```

```
    def moving_left_up(self, event):
        self.event_state = 'selected'
```

接下来是各种事件的处理方法，它们根据当前的事件状态和事件名称对元数据进行更新，由各种事件引起的事件状态漂移如图 8-20 所示。每个事件处理方法所实现的功能都很简单，这里就不再多做解释了。

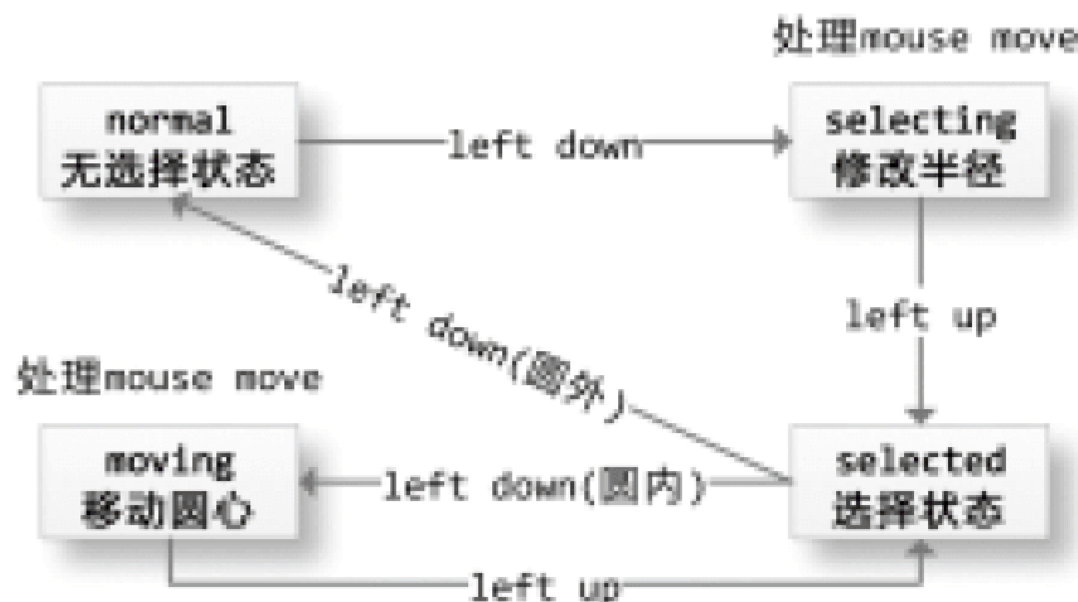


图 8-20　事件状态的漂移图

```
    def _update_selection(self):
        points = np.transpose(np.array((self.component.index.get_data(),
            self.component.value.get_data()))) ❹
        screen_points = self.component.map_screen(points) ❺
        tmp = screen_points - np.array([self.x, self.y])
        tmp **= 2
        dist = np.sum(tmp, axis=1)
        self.metadata['selections'] = dist < self.r*self.r ❻
        self.selection_update = True
```

当圆心坐标或半径发生改变时，将调用_update_selection()。它将根据当前的圆形选择区域，更新名为'selections'的元数据。❹首先，获得绘图对象 component 的数据点的坐标，points 是一个形状为(N, 2)的数组，其中 N 为绘图对象中数据点的个数。❺由于 points 数组所表示的是数据坐标系中的坐标，因此需要调用绘图对象的 map_screen()将其转换成屏幕坐标系中的坐标。❻最后选择屏幕坐标距离圆心坐标小于半径的那些数据点。'selections'元数据是一个布尔数组，它表示对应的数据点是否落入了圆形选区之内。

为了对圆形选择工具进行测试，在程序的最后编写了 CircleSelectionDemo 类。在其初始化方法__init__()中创建圆形选择工具的程序段如下所示：

```
    scatter = plot.plot(("x", "y"), type="scatter", color="blue")[0]
    scatter.tools.append( CircleSelection(scatter) )
    scatter.overlays.append( ScatterInspectorOverlay(scatter, ❼
        selection_color="red", selection_marker="circle",
        selection_outline_color = "black",
```

```
    selection_marker_size = 6) )
scatter.overlays.append( CircleSelectionOverlay(scatter) )
```

❼使用 ScatterInspectorOverlay 突出显示被选择的数据，默认使用名为'selections'的元数据表示选区。由于它在 CircleSelectionOverlay 之前被添加到 overlays 列表中，因此半透明的圆形将覆盖在被选择的数据点之上。

8.4.4 制作动画演示

使用 Chaco 或 Enable 还可以制作简单的二维动画演示。制作动画的基本方法就是使用一个定时器以一定的时间间隔重复调用某个绘图函数。由于在 Traits 和 TraitsUI 库中没有定义定时器类，因此需要从 pyface 库载入 Timer 类：

```
from enthought.pyface.timer.api import Timer
```

下面是演示正弦波移动的动画程序。



chaco_simple_line_anim.py
正弦波移动的动画演示

```
class AnimationHandler(Handler): ❶
    def init(self, info):
        super(AnimationHandler, self).init(info)
        info.object.timer = Timer(10, info.object.on_timer) ❷

    def closed(self, info, is_ok):
        super(AnimationHandler, self).closed(info, is_ok)
        info.object.timer.Stop() ❸

class AnimationPlot(HasTraits):
    plot = Instance(Plot)
    data = Instance(ArrayPlotData)
    phase = Float(0)
    traits_view = View(
        Item('plot',editor=ComponentEditor(), show_label=False),
        width=500, height=500, resizable=True, title="Plot Animation",
        handler = AnimationHandler()) ❶

    def __init__(self, **traits):
        super(AnimationPlot, self).__init__(**traits)
        data = ArrayPlotData(x=[0], y=[0])
        plot = Plot(data)
        plot.plot(("x", "y"), type="line", color="blue")
        plot.title = "sin(x)"
```

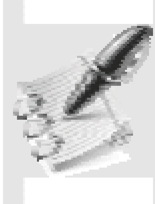
```
        self.plot = plot
        self.data = data

    def on_timer(self): ❹
        x = np.linspace(self.phase, self.phase+np.pi*4, 100)
        y = np.sin(x)
        self.phase += 0.02
        self.data["x"] = x ❺
        self.data["y"] = y
```

❶由于需要在显示界面的同时启动定时器，在界面关闭之后关闭定时器，因此定义了一个控制器类 AnimationHandler。❷它的 init()方法在界面显示时调用，我们在其中为模型对象创建定时器对象。定时器将每 10 毫秒调用一次模型对象的 on_timer()方法。❸在界面关闭之后，控制器对象的 closed()方法将被调用，我们在这里关闭定时器。如果直接在模型对象的__init__()中创建定时器，那么在界面显示之前就已经开始调用 on_timer()方法，在界面关闭之后，模型对象销毁之前，仍然会调用 on_timer()。如果模型对象的创建和销毁与界面的显示和关闭几乎同步，就可以在模型对象的__init__()中开启定时器。

❹在模型类 AnimationPlot 的 on_timer()方法中，逐渐增加 phase 属性的值，并根据当前的 phase 属性计算正弦波数据，这样就能够产生正弦波移动的效果。❺我们只需要使用新的数据更新数据源即可，当数据源中的数据改变时，Plot 对象将自动刷新。

使用同样的方法，我们也可以直接使用 Component 类制作动画演示程序。下面的程序是一个很酷的三维点阵变形动画演示。它在将随机产生的三维曲面上的点进行三维旋转和投影之后，在 Compoent 组件上使用 Kiva 的绘图函数快速绘制出来。此程序留给读者自行研究，这里就不再进行详细解释了。图 8-21 是此程序绘制的一些三维点阵图形。



enable_morph3d_anim.py
三维点阵变形动画

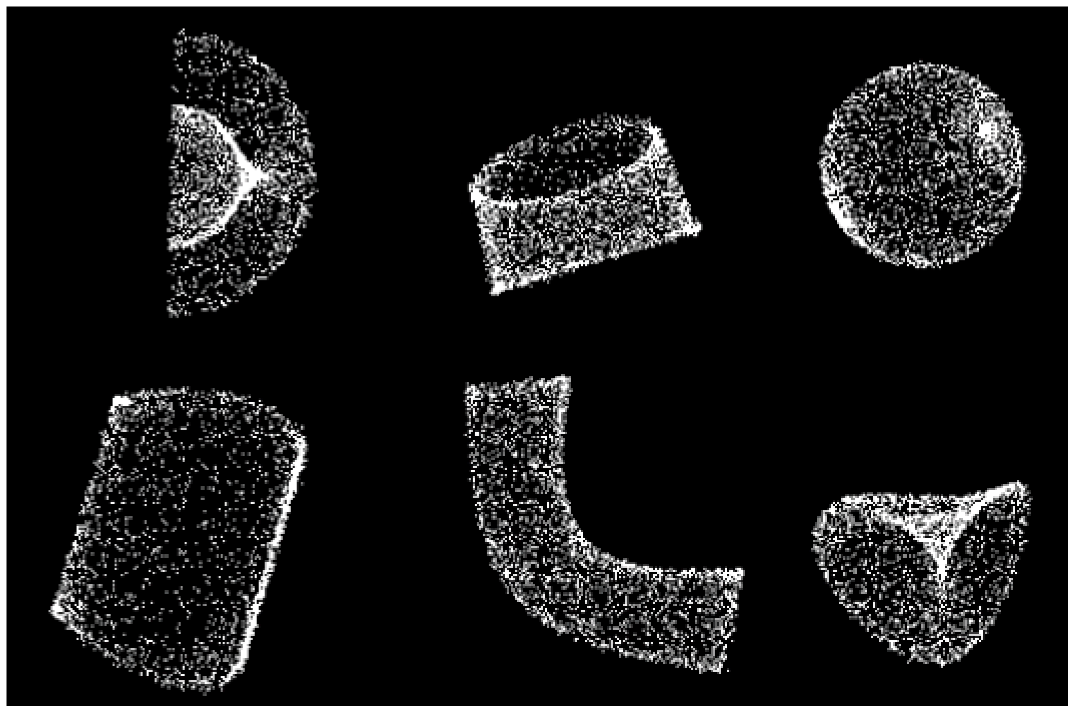


图 8-21　三维点阵变形动画中的一些图形截图

第 9 章

TVTK——数据的三维可视化

VTK 是一套功能十分强大的三维的数据可视化库，它使用 C++编写而成，其中包含了近千个类。它在 Python 下有标准的扩展库，不过由于其 Python 扩展库的 API 和 C++的 API 相同，无法体现出 Python 作为动态语言的优势。因此，Enthought 公司开发了一套名为 TVTK 的扩展库来对 VTK 进行封装，它提供了 Python 风格的 API，并支持 Trait 属性和 NumPy 数组。本章将以 TVTK 的 API 为例介绍如何在 Python 中使用 VTK 实现数据的三维可视化。

由于 TVTK 库十分庞大，为了方便用户查询文档，TVTK 库提供了一个显示 TVTK 文档的工具。读者可以通过下面的语句运行它：

```
>>> from enthought.tvtk.tools import tvtk_doc
>>> tvtk_doc.main()
```

TVTK 库提供的工具并不太好用，因此本书为读者提供了一个更方便的 TVTK 文档查询工具，其界面如图 9-1 所示。

tvtk_class_doc.py
TVTK 文档查询工具

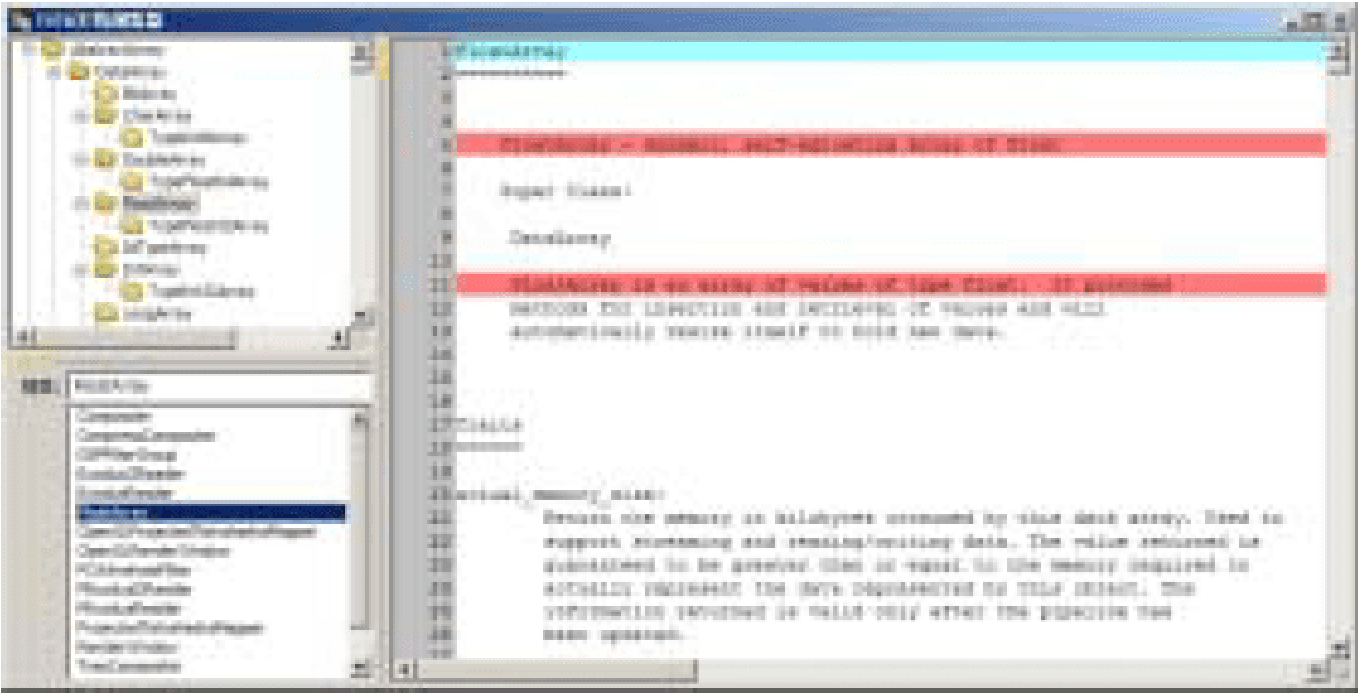


图 9-1　TVTK 文档查询工具

第一次运行此文档工具时，它将对 TVTK 库中所有的类进行扫描，并将类的继承关系和文档全部保存在“tvtk_classes.cache”中，这个过程可能需要等待较长的时间。

界面的左上部分使用一个树型控件来显示 TVTK 中各个类之间的继承关系。在中间的文本框中输入搜索文本，下方的列表框中会实时显示搜索结果。输入全小写字母，进行忽略大小写的搜索，而输入带大写字母的文本则会进行精确搜索。

9.1 流水线(Pipeline)

VTK 是一个十分复杂的系统，为了方便用户使用，它使用流水线技术，将 VTK 中的各个对象串联起来。每个对象只需要实现相对简单的任务，整个流水线则能够根据用户的需求实现十分复杂的数据可视化处理。

9.1.1 显示圆锥

作为第一例子，让我们首先看一个显示圆锥的小程序，它的运行效果如图 9-2 所示。



tvtk_simple_cone.py
用 TVTK 快速显示圆锥

```
from enthought.tvtk.api import tvtk ❶

# 创建一个圆锥数据源，并且同时设置其高度、底面半径和底面圆的分辨率(用 36 边形近似)
cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36) ❷
# 使用 PolyDataMapper 将数据转换为图形数据
m = tvtk.PolyDataMapper(input = cs.output) ❸
# 创建一个 Actor
a = tvtk.Actor(mapper=m) ❹
# 创建一个 Renderer，将 Actor 添加进去
ren = tvtk.Renderer(background=(1, 1, 1)) ❺
ren.add_actor(a)
# 创建一个 RenderWindow(窗口)，将 Renderer 添加进去
rw = tvtk.RenderWindow(size=(300,300)) ❻
rw.add_renderer(ren)
# 创建一个 RenderWindowInteractor(窗口的交互工具)
rwi = tvtk.RenderWindowInteractor(render_window=rw) ❼
# 开启交互
rwi.initialize()
rwi.start()
```



图 9-2　使用 TVTK 绘制简单的圆锥

❶首先从 TVTK 库中载入 tvtk 对象，它可以帮助我们创建 TVTK 库中的各种对象。❷创建了一个 ConeSource 对象，它是计算圆锥形状的数据源对象。在 TVTK 中，所有类都从 HasTraits 继承，因此可以在创建对象的同时，使用关键字参数直接设置各个 Trait 属性的值。在这个例子中，同时设置了圆锥的高度、底面半径和底面圆的边数[①]等属性。

tvtk 对象是什么

事实上，我们载入的 tvtk 并不是一个模块，而是某个类的实例。之所以如此设计，是因为 VTK 库有近千个类，而 TVTK 对这些类进行了封装，如果一次性载入这么多类，会极大地影响库的载入速度。

我们载入的 tvtk 虽然是某个实例对象，但是用起来却和模块一样：

- 通过它可以使用所有的 TVTK 类。
- 它不需要载入近千个 TVTK 类就能支持类名的自动补全。
- 只有在真正使用时，TVTK 类才会载入。

所有对 VTK 进行封装的类全部保存在“tvtk_classes.zip”文件中，而 tvtk 对象的类则在此压缩文件的“tvtk_helper.py”中定义。对于 VTK 中的每个类，tvtk 对象都有一个同名的属性与之对应。

可以使用 print_traits()显示 ConeSource 对象的所有 Trait 属性，为了节省篇幅，这里只挑选了其中的几个属性：



在 IPython 中运行此程序之后，将无法关闭显示圆锥的窗口，为了关闭它，请直接关闭 IPython 的窗口。

① 圆锥的底面实际上是一个正多边形。

```
>>> run tvtk_simple_cone.py
>>> cs.print_traits()
angle:                          18.43494882292201
center:                         array([ 0.,  0.,  0.])
class_name:                     'vtkConeSource'
direction:                      array([ 1.,  0.,  0.])
height:                         3.0
radius:                         1.0
resolution:                     36
[[省略]]
```


为了将原始数据转换为屏幕上的一幅图像，需要经过许多处理步骤。这些步骤由众多的VTK对象分步实现，就好像生产线上加工零件一样，每位工人都负责一部分工作，整条生产线就能将原材料制作成产品。在VTK中，这种在各个对象之间协调完成工作的过程被称为流水线(Pipeline)。

原始数据被加工成图像要经过两条流水线：

- 可视化流水线(Visualization Pipeline)：它的工作是将原始数据加工成图形数据。一般来说，我们需要进行可视化展示的数据本身并不是图形数据，例如它可能是某个零件内部各个部分的温度，或者是流体中各个坐标点上的速度等。
- 图形流水线(Graphics Pipeline)：它的工作是将图形数据加工为我们所看到的图像。可视化流水线所产生的图形数据通常是三维空间的数据，图形流水线将这些三维数据加工成能在二维屏幕上显示的图像。

❸映射器(Mapper)是可视化流水线的终点、图形流水线的起点，它的各种派生类能将众多的数据映射为图形数据以供图形流水线加工。在本例中，ConeSource对象输出一个描述圆锥的顶点和面的PolyData对象，然后PolyData对象通过PolyDataMapper映射器转换为图形数据。因此在本例中，可视化流水线由ConeSource对象和PolyDataMapper对象组成。

可视化流水线中的对象经由input和output属性连接起来。例如ConeSource对象的output属性是一个PolyData对象，它也是PolyDataMapper对象的input属性。我们可以认为ConeSource对象产生了一个PolyData对象，并转交给PolyDataMapper对象进行处理：

```
>>> cs.output
<tvtk_classes.poly_data.PolyData object at 0x048EBD80>
>>> m.input
<tvtk_classes.poly_data.PolyData object at 0x048EBD80>
```

然后图形数据再依次通过Actor、Renderer最终在RenderWindow中显示出来，这一部分就是图形流水线。❹Actor对象代表场景中的一个实体。它的mapper属性是表示图形数据的PolyDataMapper对象，Actor对象还有许多属性可以控制实体的位置、方向、大小等：

```
>>> a.mapper is m
```

```
True
>>> a.scale # Actor 对象的 scale 属性表示各个轴的缩放比例
array([ 1.,  1.,  1.])
```

❺Renderer 对象表示三维场景，它可以包括多个 Actor 对象，这些 Actor 对象都保存在 actors 列表属性中。在本例中，它只包括一个显示圆锥的 Actor 对象：

```
>>> ren.actors
['<tvtk_classes.actor.Actor object at 0x04D75720>']
```

❻RenderWindow 对象表示包含场景的窗口，它可以同时包含多个场景。在本例中，它只有一个 Renderer 对象：

```
>>> rw.renderers
['<tvtk_classes.renderer.Renderer object at 0x04F01600>']
```

❼RenderWindowInteractor 对象为图形窗口提供一些用户交互功能，例如平移、旋转和缩放。这些交互式操作并不改变场景中各个实体(Actor 对象)或图形数据的属性，它们只是修改场景中照相机(Camera)的设置，以便从不同的角度和距离观察场景中的实体。

9.1.2 用 ivtk 观察流水线

为了方便对流水线进行观察和操作，TVTK 库为我们提供了一个很方便的工具——ivtk。使用它可以交互式地对各种 TVTK 对象的属性进行编辑，下面是使用 ivtk 显示圆锥的程序，运行画面如图 9-3 所示。



tvtk_ivtk_cone.py
带流水线浏览器和 Python 命令行的 ivtk 界面

```
from enthought.tvtk.api import tvtk

# 载入 ivtk 所需要的对象
from enthought.tvtk.tools import ivtk
from enthought.pyface.api import GUI

cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36)
m = tvtk.PolyDataMapper(input = cs.output)
a = tvtk.Actor(mapper=m) ❶

# 创建一个 GUI 对象，和一个带 Crust(Python shell)的 ivtk 窗口
gui = GUI()
window = ivtk.IVTKWithCrustAndBrowser(size=(800,600)) ❷
window.open()
window.scene.add_actor( a ) ❸ # 将圆锥的 actor 添加到窗口的场景中
```

```
gui.start_event_loop()
```

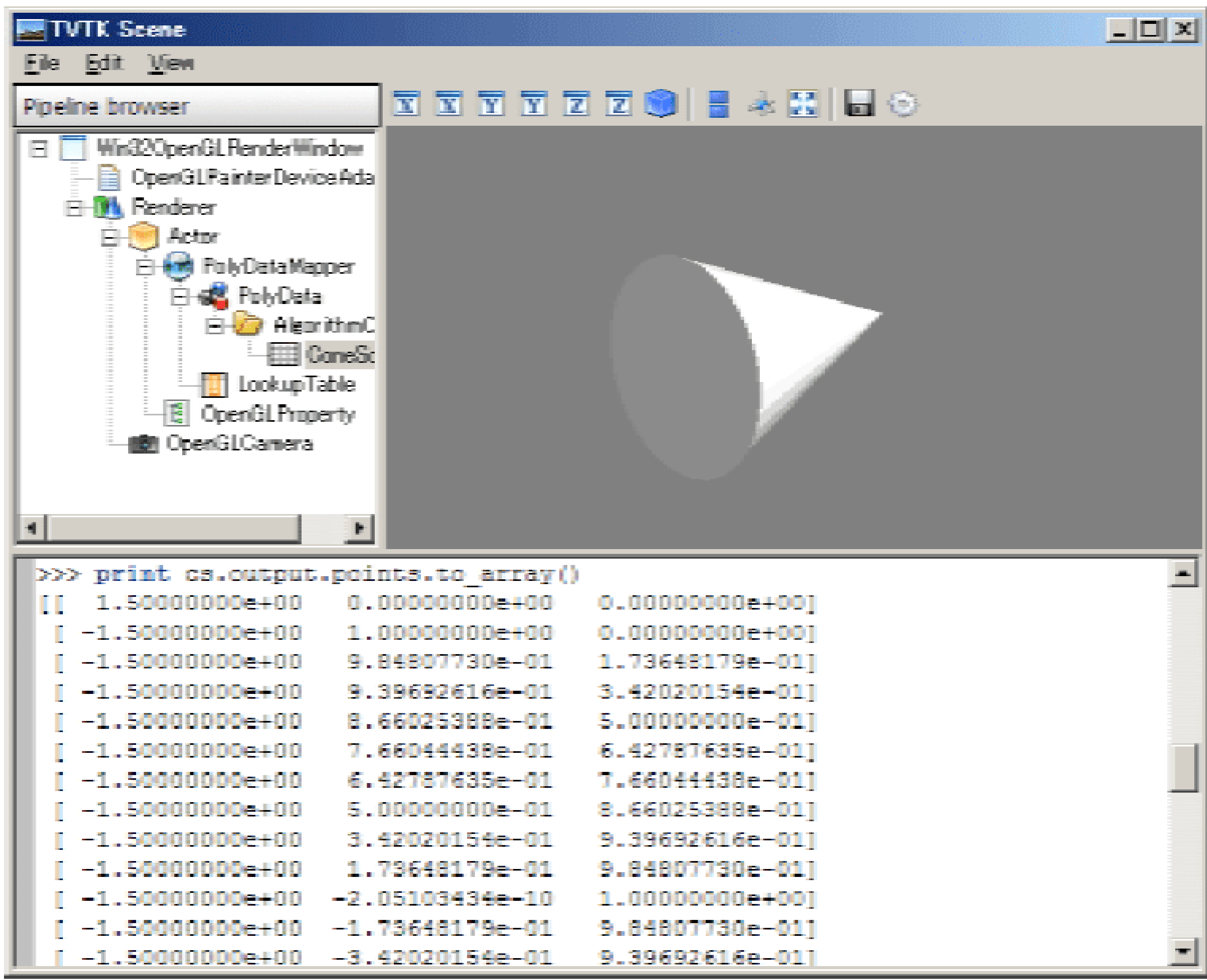


图 9-3　带流水线浏览器和 Python 命令行的 ivtk 界面

❶在创建了表示圆锥的 Actor 对象之后，❷创建并显示了一个 ivtk 窗口，❸调用窗口对象表示场景的 scene 属性的 add_actor()方法将 Actor 对象添加到场景中。关于这段程序的详细解释已经超出了本书的范围，请感兴趣的读者自行查看各个函数的文档和源代码进行分析。下面让我们看看 ivtk 窗口的各个组成部分：

- 场景：用于显示可视化的结果，这里只显示了一个圆锥。
- 场景工具条：位于场景的上方，主要提供了各种视角、全屏显示、保存图像等功能。
- 流水线浏览器：场景的左上方是一个表示流水线的树状控件。从子节点(ConeSource)开始逐步向上层直到根节点(Render)，是整个显示圆锥的流水线。
- Python 命令行：界面下方提供了一个 Python 命令行，在其中可以访问程序中的全局变量，可方便用户直接输入命令来操作各个对象。例如图中显示了 ConeSource 对象所输出的 PolyData 对象的 points 属性，即构成圆锥图形的各个顶点的三维坐标。

流水线浏览器中显示的各个对象的类都从 HasTraits 继承，因此它们可以提供一个用户界面，交互式地修改其 Trait 属性。图 9-4 是双击流水线中的 ConeSource 对象之后弹出的属性编辑界面。通过此界面可以直接修改 height、radius、resolution 等属性，并且修改之后场景中的圆锥会根据最新的属性值立即更新显示。


双击 ConeSource 之后弹出的窗口可能很小，需要手动调整其大小。

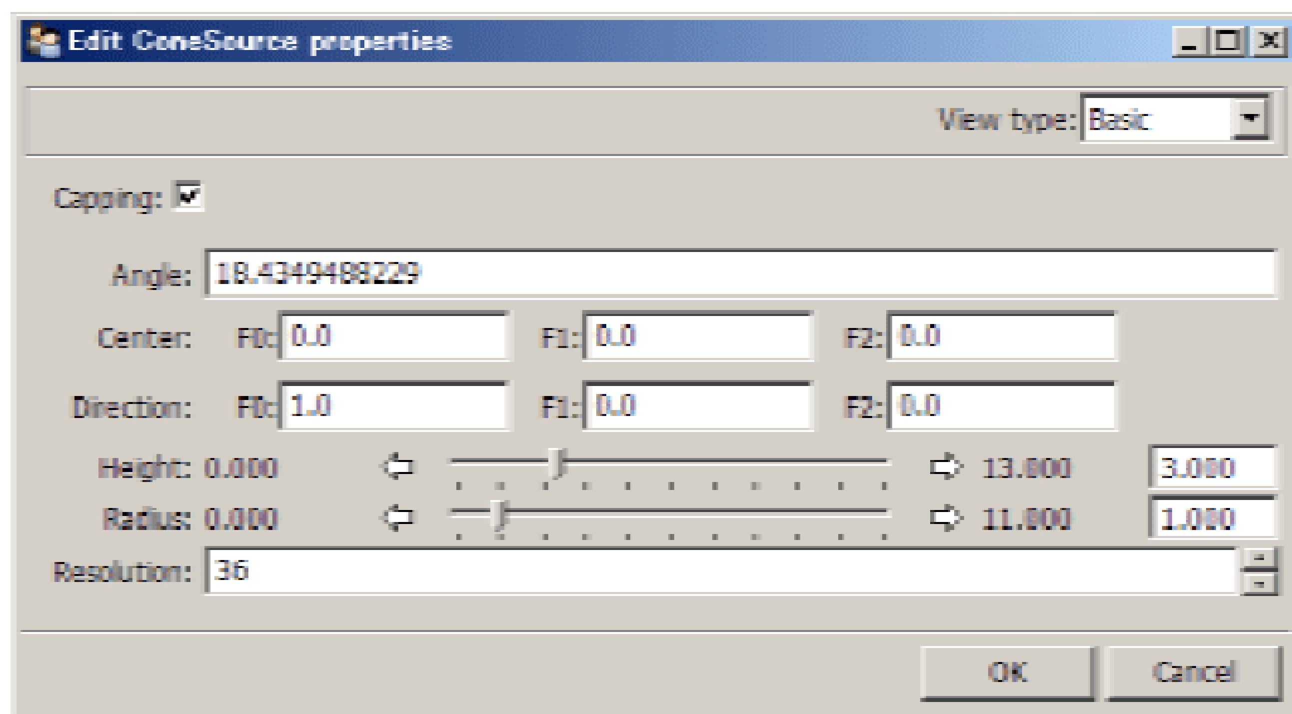


图 9-4　用于编辑 ConeSource 对象属性的对话框

1. 照相机

在 ivtk 窗口左侧的流水线浏览器中可以找到场景中的照相机对象——OpenGLCamera，双击它将弹出如图 9-5 所示的编辑照相机对象属性的对话框。

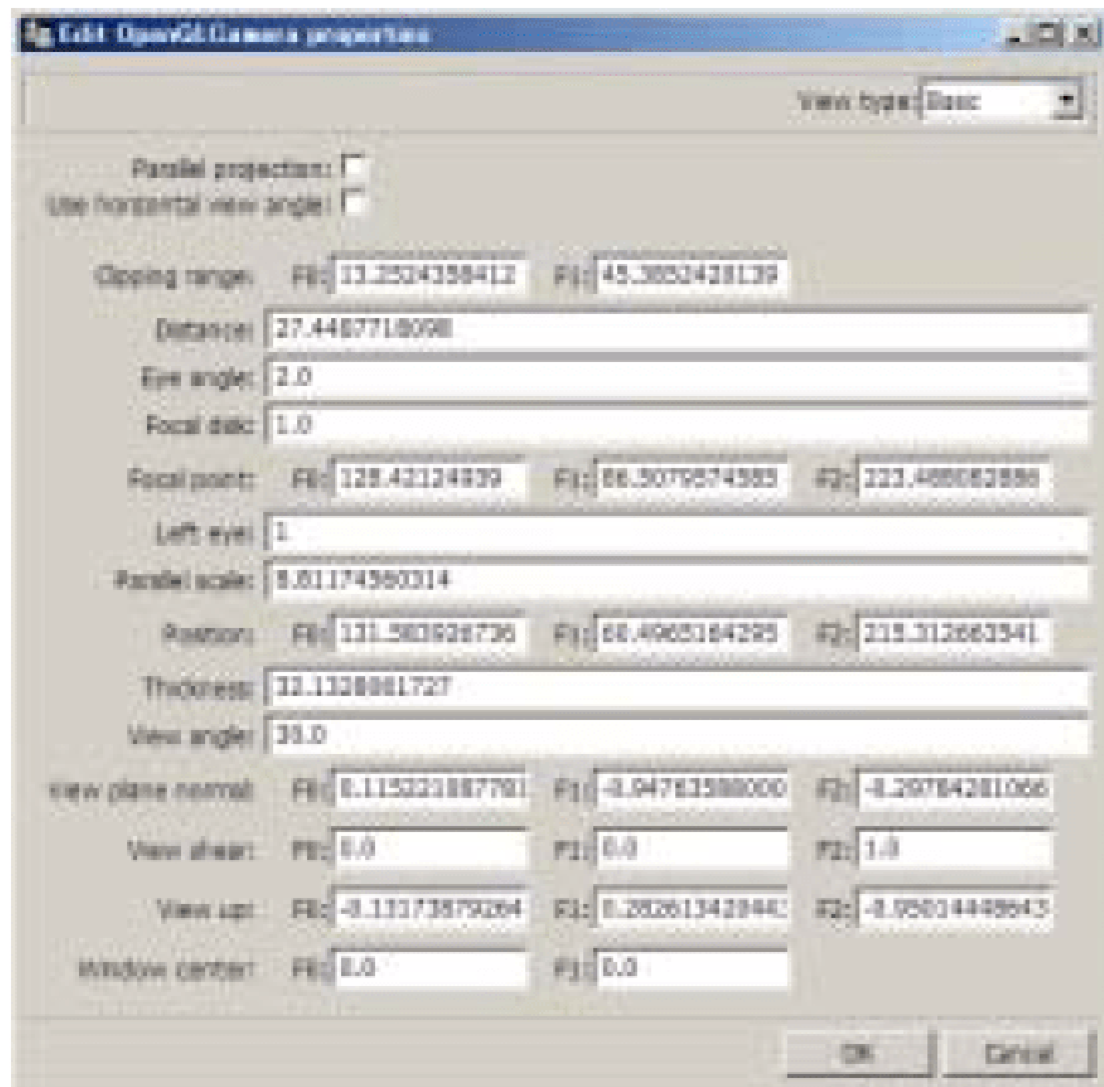


图 9-5　用于编辑照相机属性的对话框

也可以使用下面的程序从窗口对象 window 获得照相机对象，然后查看或修改它的属性：

```
>>> camera = window.scene.renderer.active_camera
>>> camera.clipping_rage
array([ 20.46912341,  51.21854284])
>>> camera.view_up = 0,1,0
>>> camera.edit_traits() # 显示编辑照相机属性的对话框
```

下面列出照相机对象的一些常用属性：

- clipping_plane：它有两个元素，分别表示照相机到近、远两个裁剪平面的距离。在这两个平面之外的对象将不会显示。
- position：照相机在三维空间中的坐标。
- focal_point：照相机所聚焦的焦点坐标。
- view_up：照相机的上方向矢量。
- parallel_projection：True 表示采用平行透视，即在三维场景中平行的直线在屏幕上也是平行的。

这些属性虽然可以完全控制照相机的位置和方向，但是实际操作起来并不方便。如果已经将照相机的焦点固定在某个位置，可以调用照相机对象的下面两个方法，在以焦点为原点的球面坐标系中对照相机进行操作。它们保持照相机的 view_up 属性不变。

- azimuth(angle)：沿着纬度线旋转指定角度，即水平旋转，改变其经度。
- elevation(angle)：沿着经度线方向旋转指定角度，即垂直旋转，改变其纬度。

2. 光源

在 ivtk 窗口中，单击场景上方工具栏中最后一个齿轮形状的图标，将打开如图 9-6 所示的编辑场景和光源的对话框。在此对话框中可以添加和删除光源，还可以修改它们的一些属性。

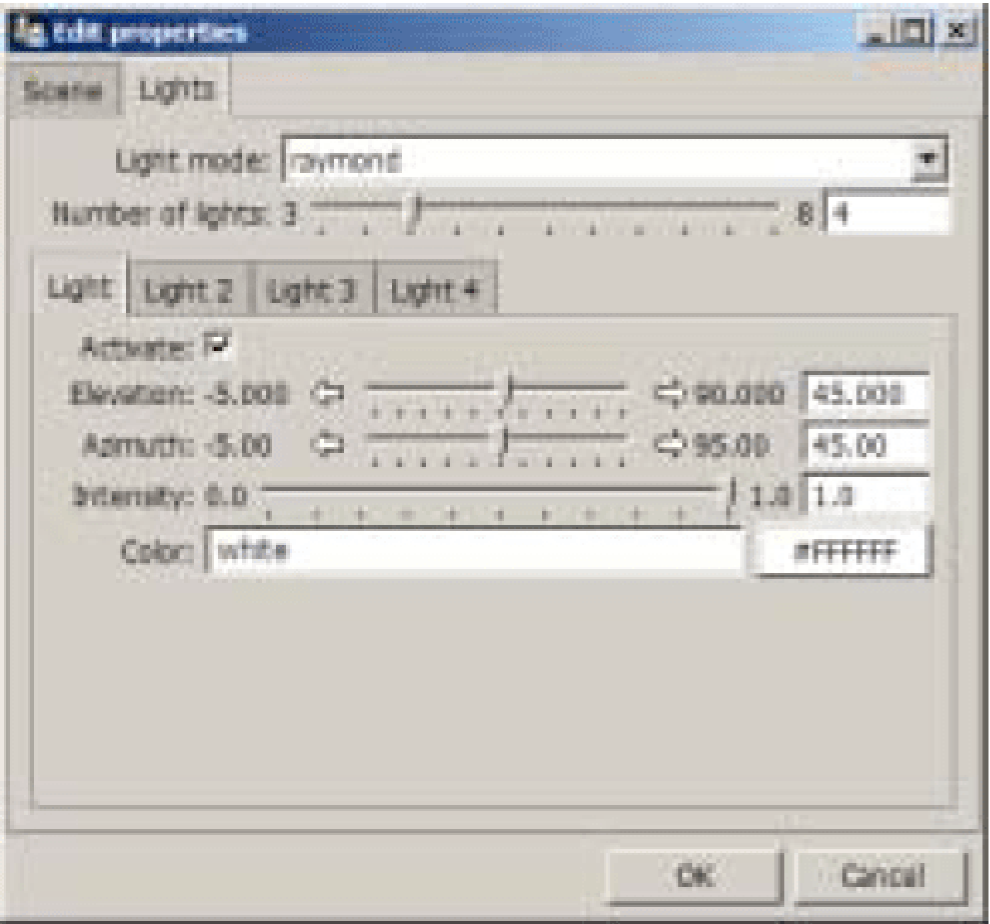


图 9-6　设置场景和光源的对话框

场景中的光源可以通过 Renderer 对象的 lights 属性获得，它是一个光源对象的列表。Renderer 对象还有 add_light()和 remove_light()等方法，用来添加或删除光源对象。

```
>>> lights = window.scene.renderer.lights
>>> lights[0].edit_traits() # 显示编辑光源属性的对话框
```

下面的程序在照相机所在位置添加一个红色的光源，它的照射方向和照相机的方向相同，

朝向 focal_point 点。如果设置光源对象的 positional 属性为 True，它将变成一个探照灯光源，这时照射方向有效；并且可以通过 cone_angle 属性设置探照灯的光锥角度，如果光锥为 180°，它就是无方向光源。

```
>>> camera = window.scene.renderer.active_camera
>>> light = tvtk.Light(color=(1,0,0))
>>> light.position=camera.position
>>> light.focal_point=camera.focal_point
>>> window.scene.renderer.add_light(light)
```

3. 实体

Actor 对象表示场景中的实体，在圆锥的流水线浏览器中，我们可以看到一个表示圆锥的 Actor 对象。双击它可以打开如图 9-7 所示的对话框。也可以使用下面的代码打开此对话框：

```
>>> a.edit_traits() # a 是表示圆锥的 Actor 对象
>>> window.scene.renderer.actors[0].edit_traits()
```



在 IPython 打开的属性编辑对话框中进行修改之后，需要激活一下场景窗口才能让它进行重绘。

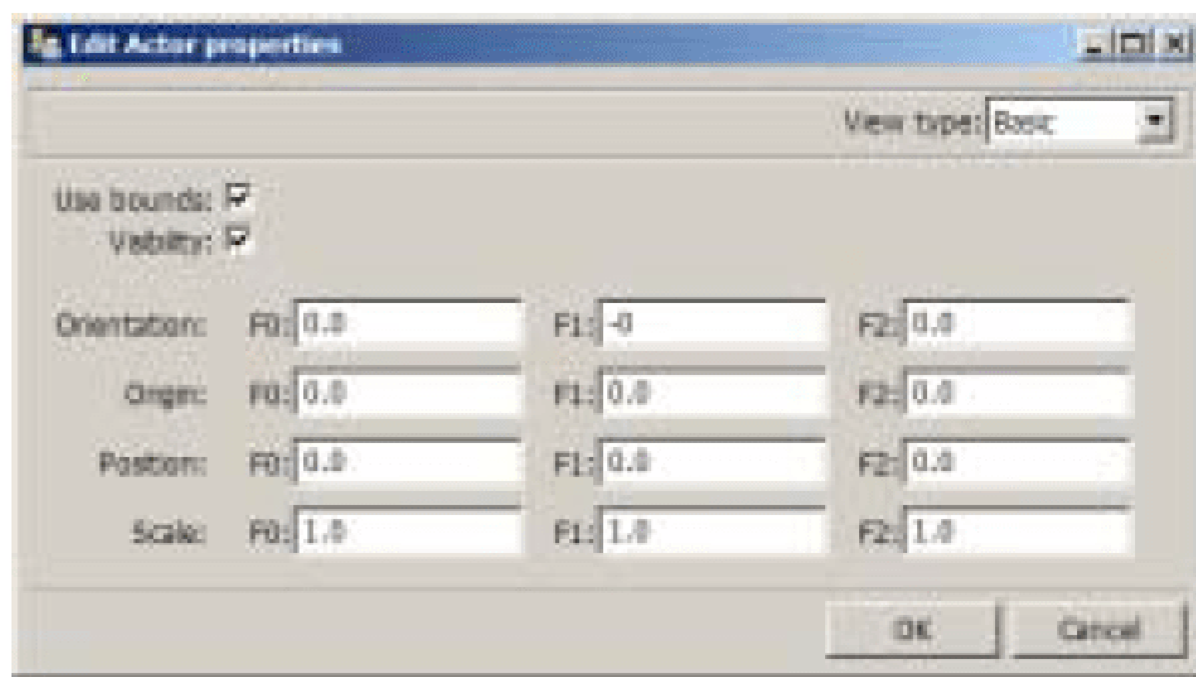


图 9-7　Actor 对象的属性编辑对话框

在此对话框中可以编辑 Actor 对象的 origin、position、orientation 和 scale 属性，还可以修改场景中实体的位置、方向及大小。这 4 个属性通过一系列复杂的计算可得到一个 4 × 4 的三维空间的变换矩阵。其变换步骤如下：

- 以 origin 为中心，使用 scale 对物体在三个轴上进行缩放。
- 以 origin 为中心，使用 rotate 对物体在三个轴上进行旋转，旋转的顺序是 Y 轴→X 轴→Z 轴。
- 将物体放到 position 处。

我们通过下面的例子理解坐标变换的步骤。首先运行如下程序，在场景中添加一个坐标轴：

```
>>> run tvtk_ivtk_cone.py
>>> axe = tvtk.AxesActor(total_length=(3,3,3)) # 在场景中添加坐标轴
>>> window.scene.add_actor( axe )
```

可以看到屏幕的横轴方向是 X 轴，纵轴方向是 Y 轴，从屏幕里往外是 Z 轴方向。 整个圆锥的长度为 3，它的底面在 X = -1.5 的平面之上。双击流水线对话框中表示圆锥的 Actor，打开如图 9-7 所示的对话框。

将 origin 修改为(-1.5, 0, 0)，这样将以圆锥的底面圆心为中心进行缩放和旋转。依次按照图 9-8 的顺序修改各个属性的值。对话框中的“F0”、“F1”和“F2”等标签分别表示 X 轴、Y 轴和 Z 轴的分量。

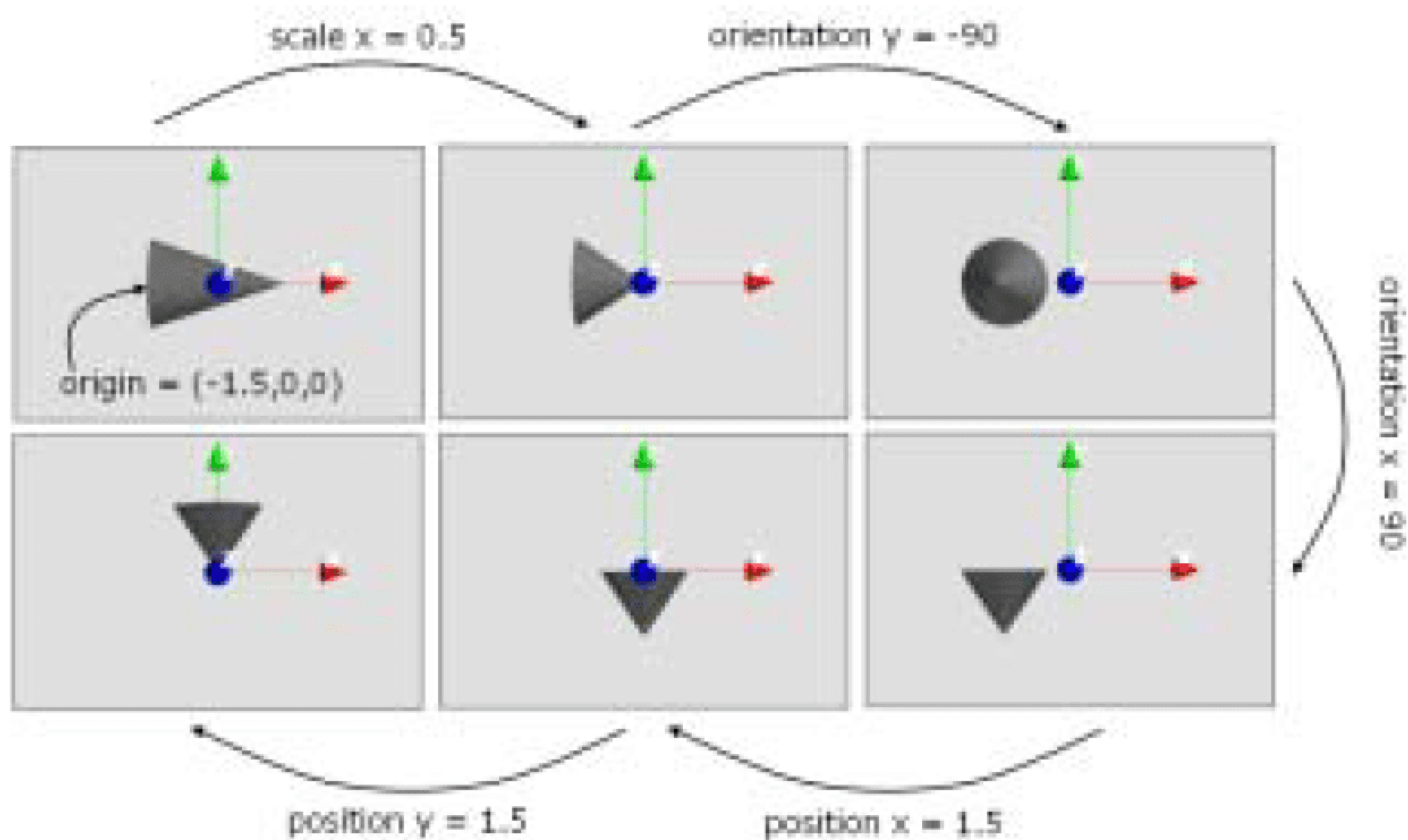


图 9-8　依次修改圆锥的 scale、orientation 和 position 属性

请读者仔细观察每两幅图之间的变化，分析并理解前面的坐标变换步骤。旋转的正方向按照右手法则来决定：右手握拳，并伸出大拇指让它指向某个轴的正方向，其余四指的方向为绕此轴旋转的正方向。

Actor 对象的 property 属性是一个 OpenGLProperty 对象，它包含了对实体进行着色时所使用的各种配置，例如 color 属性是实体的颜色、opacity 属性是实体的不透明度。输入下面的语句可以打开编辑这些属性的对话框：

```
>>> a.property.edit_traits() # a 是表示圆锥的 Actor 对象
```

由于 OpenGLProperty 对象的属性太多，这里不一一进行介绍。在后面的实例中用到时再对其进行说明。

9.2 数据集(Dataset)

数据可视化的第一步是用合适的数据结构表示数据，VTK 提供了多种表示不同种类数据的数据集(Dataset)。数据集包括点(Point)和数据(Data)两个部分。点之间可以是连接的或非连接的，多个相关的点组成单元(Cell)，而点之间的连接可以是显式或隐式的。数据可以是标量或矢量，数据可以属于点或单元。下面让我们通过一些实例逐步理解数据集的构造。

为了帮助读者更形象地了解数据集的结构，我们使用 Mayavi 将数据集的结构绘制成三维图。请读者在学习的过程中运行这些程序，以加深对数据集的理解。在学习第 10 章时，也可以将这些程序作为实例，来了解 Mayavi 的一些高级用法。



figure_*.py
使用 Mayavi 绘制数据集的结构

9.2.1 ImageData

最容易理解的数据集是 ImageData，它是表示二维或三维图像的数据结构。我们可以简单地将其理解为二维或三维数组。数组中存放的是数据，由于点位于正交、等间距的网格之上，因此不需要给出点的坐标，而点之间的连接关系也由它们在数组中的位置决定，因此连接也是隐式的。



tvtk_imagedata.py
ImageData 对象的演示程序

下面的代码创建了一个 ImageData 对象，并且设置了它的 spacing、origin 和 dimensions 属性：

```
>>> img = tvtk.ImageData(spacing=(0.1,0.1,0.1), origin=(0.1,0.2,0.3), dimensions=(3,4,5))
```

origin 属性为三维网格数据的起点坐标，spacing 属性为三维网格在 X 轴、Y 轴和 Z 轴上的间距，dimensions 属性为 X 轴、Y 轴和 Z 轴上的网格数。img.get_point(n)可以获得网格中第 n 个点的坐标值，n 为点的序号，起始点的序号为 0，序号依次沿着 X 轴、Y 轴和 Z 轴递增。下面的程序输出前 6 个点的坐标：

```
>>> for n in range(6):
...:     print "%.1f, %.1f, %.1f" % img.get_point(n)
0.1, 0.2, 0.3
0.2, 0.2, 0.3
0.3, 0.2, 0.3
0.1, 0.3, 0.3
```

```
0.2, 0.3, 0.3
0.3, 0.3, 0.3
```

与每个点对应的数据都保存在 point_data 属性中，它是一个 PointData 对象：

```
>>> img.point_data
<tvtk_classes.point_data.PointData object at 0x06022270>
```

PointData 对象可以保存多组数据，它的 scalars 属性是 VTK 库中的一个数组对象，用来保存与每个点对应的标量值。当将一个 NumPy 数组赋值给它时，TVTK 将自动创建 VTK 中相应的数组对象，并保存 NumPy 数组的内容。例如下面的程序运行之后，scalars 属性从 None 变成了一个 DoubleArray 数组。程序为每个点添加了一个与其序号相同的标量值：

```
>>> import numpy as np
>>> img.point_data.scalars # 没有数据
None
>>> img.point_data.scalars = np.arange(0.0, img.number_of_points)
>>> img.point_data.scalars
[0.0, ..., 59.0], length = 60
>>> type(img.point_data.scalars)
<class 'tvtk_classes.double_array.DoubleArray'>
```

DoubleArray 数组只能以整数为下标，不支持切片以及其他高级下标运算。可以通过它的 to_array()方法获得与其共享数据存储空间的 NumPy 数组，通过此 NumPy 数组可以进行更高级的数据存取操作：

```
>>> a = img.point_data.scalars.to_array()
>>> a
array([  0.,   1.,   2.,   3.,  ...])
>>> a[:2] = 10, 11
>>> img.point_data.scalars[0]
10.0
>>> img.point_data.scalars[1]
11.0
```

VTK 的数组对象有许多属性，可以用 print_traits()方法查看各个属性的值，例如 number_of_tuples 属性表示数组的长度：

```
>>> img.point_data.scalars.number_of_tuples
60
```

每个 VTK 数组都有一个 name 属性保存其名称，下面的语句将数组的名称设置为'scalars'：

```
>>> img.point_data.scalars.name = 'scalars'
```

PointData 对象可以保存多个数组，它所包含的数组的个数可以通过 number_of_arrays 属性获得，可以通过 add_array()方法添加新的数组对象，通过 remove_array()方法删除数组对象，通过 get_array()和 get_array_name()方法分别获得数组对象及其名称。下面我们演示这些方法的用法。首先创建一个 TVTK 的数组对象，并调用其 from_array()方法，通过 NumPy 数组设置其内容，数组的长度为 ImageData 对象的点数：

```
>>> data = tvtk.DoubleArray() # 创建一个空的 DoubleArray 数组
>>> data.from_array(np.zeros(img.number_of_points))
```

接下来将 TVTK 数组对象的名称设置为"zerodata"：

```
>>> data.name = "zerodata"
```

然后调用 PointData 对象的 add_array()方法，将创建的数组添加到 PointData 对象中。当 PointData 对象已经有相同名称的数组时，将会覆盖原来的数组。add_array()方法的返回值是新数组的序号，可以通过此序号获得数组或数组名：

```
>>> img.point_data.add_array(data)
1
>>> img.point_data.get_array(1) # 获得第 1 个数组
[0.0, ..., 0.0], length = 60
>>> img.point_data.get_array_name(1) # 获得第 1 个数组的名称
'zerodata'
>>> img.point_data.get_array(0) # 获得第 0 个数组
[10.0, ..., 59.0], length = 60
>>> img.point_data.get_array_name(0) # 获得第 0 个数组的名称
'scalars'
```

最后，使用 remove_array()方法通过数组名删除数组：

```
>>> img.point_data.remove_array("zerodata") # 删除名为"zerodata"的数组
>>> img.point_data.number_of_arrays
1
```

可以用上述方法对 PointData 对象中的多个数组进行管理，同时为了方便使用，它的 scalars 属性也可用于保存数组。与每个点对应的值除了可以是标量之外，还可以是矢量或张量(矩阵)，矢量数组可以使用 vectors 属性保存，而张量数组可以使用 tensors 属性保存。下面的程序将一个形状为(N,3)的 NumPy 数组赋值给 vectors 属性，其中 N 为点数，这样 ImageData 对象中的每个点都对应一个三维空间中的矢量：

```
>>> vectors = np.arange(0.0, img.number_of_points*3).reshape(-1, 3)
>>> img.point_data.vectors = vectors
>>> img.point_data.vectors
```

```
[(0.0, 1.0, 2.0), ..., (177.0, 178.0, 179.0)], length = 60
>>> type(img.point_data.vectors)
<class 'tvtk_classes.double_array.DoubleArray'>
>>> img.point_data.vectors[0]
(0.0, 1.0, 2.0)
```

我们看到：创建的仍然是一个 DoubleArray 对象，但它是二维数组。number_of_tuples 属性获得其第 0 轴的长度，而 number_of_components 属性获得其第 1 轴的长度：

```
>>> img.point_data.vectors.number_of_tuples
60
>>> img.point_data.vectors.number_of_components
3
```

同样，直接使用 DoubleArray 对象的方法或属性对其进行操作比较繁琐，因此建议读者仍然使用 to_array()方法在将其转换为 NumPy 数组之后再进行操作。

在 ImageData 对象中，点的坐标是通过 spacing、origin 和 dimensions 等属性隐式定义的。与之类似，点和单元(Cell)之间的关系也是隐式定义的。单元和点之间的关系如图 9-9 所示(见文前彩插)。单元是由 8 个邻近的点构成的立方体，图中使用半透明灰色立方体标识出第 0 个单元。

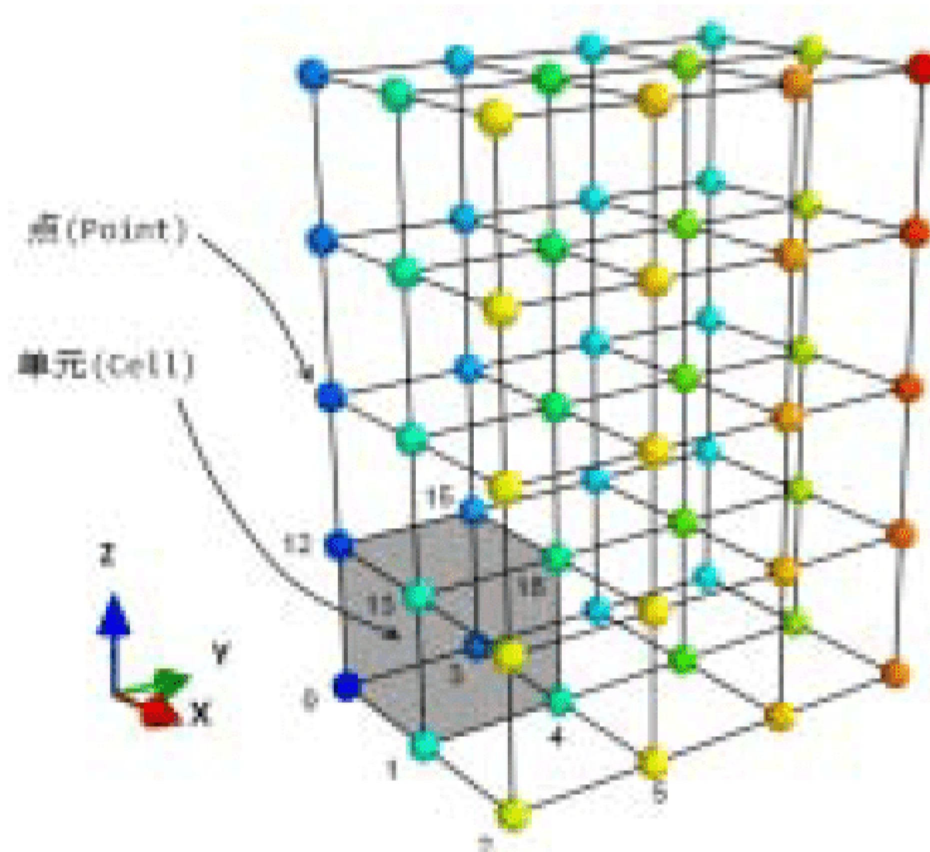


图 9-9　单元(Cell)和点(Point)之间的关系

通过 get_cell()方法可以获得表示单元的 Voxel(体素)对象，它的 number_of_points、number_of_edges 和 number_of_faces 等属性分别是构成 Voxel 对象的点数、边数及面数：

```
>>> cell = img.get_cell(0)
>>> cell
<tvtk_classes.voxel.Voxel object at 0x05E05E40>
>>> cell.number_of_points
8
```

```
>>> cell.number_of_edges
12
>>> cell.number_of_faces
6
```

point_ids 属性可以获得构成 Voxel 对象的点的序号列表，而 points 属性则获得构成 Voxel 对象的点的坐标：

```
>>> cell.point_ids
[0, 1, 3, 4, 12, 13, 15, 16]
>>> cell.points
[(0.10000000000000001, 0.20000000000000001, 0.29999999999999999), ...]
```

ImageData 对象的 number_of_cells 属性是数据集中的单元数，也就是图 9-9 中小立方体的数目：

```
>>> img.number_of_cells
24
```

数据集提供了许多获取点和单元之间关系的方法。例如，get_point_cells()获得包含某个点的所有单元的序号，而 get_cell_points()则获得某个单元包含的所有点的序号。由于这两个方法将结果写入一个 IdList 对象中，因此我们需要先创建一个空的 IdList 对象以保存结果：

```
>>> a = tvtk.IdList()
>>> img.get_point_cells(3, a)
>>> a  # 序号为 2 和 0 的单元包含序号为 3 的点
[2, 0]
>>> img.get_cell_points(0, a)
>>> a # 和 cell.point_ids 的值相同
[0, 1, 3, 4, 12, 13, 15, 16]
```

IdList 是 VTK 中管理序号的列表对象，它和 Python 的标准列表一样支持 append()和 extend()方法，并且可以通过 from_array()将列表或数组转换为 IdList 对象：

```
>>> a = tvtk.IdList()
>>> a.from_array([1,2,3])
>>> a.append(4)
>>> a.extend([5,6])
>>> a
[1, 2, 3, 4, 5, 6]
```

与每个单元对应的数据都保存在 cell_data 属性中，它是一个 CellData 对象，其用法和 PointData 对象类似，这里就不再多做介绍了。

```
>>> img.cell_data
<tvtk_classes.cell_data.CellData object at 0x0139C630>
```

9.2.2　RectilinearGrid

ImageData 是最简单的数据集，它的所有点都在一个等间距的三维网格之上，因此只需要起始坐标、网格大小以及网格间距等信息就可以计算出网格上所有点的坐标。如果要表示间距不均匀的网格，那么可以使用 RectilinearGrid 数据集。下面的程序创建如图 9-10 所示的网格(见文前彩插)：

Tvtk_rectilineargrid.py
RectilinearGrid 对象的演示程序

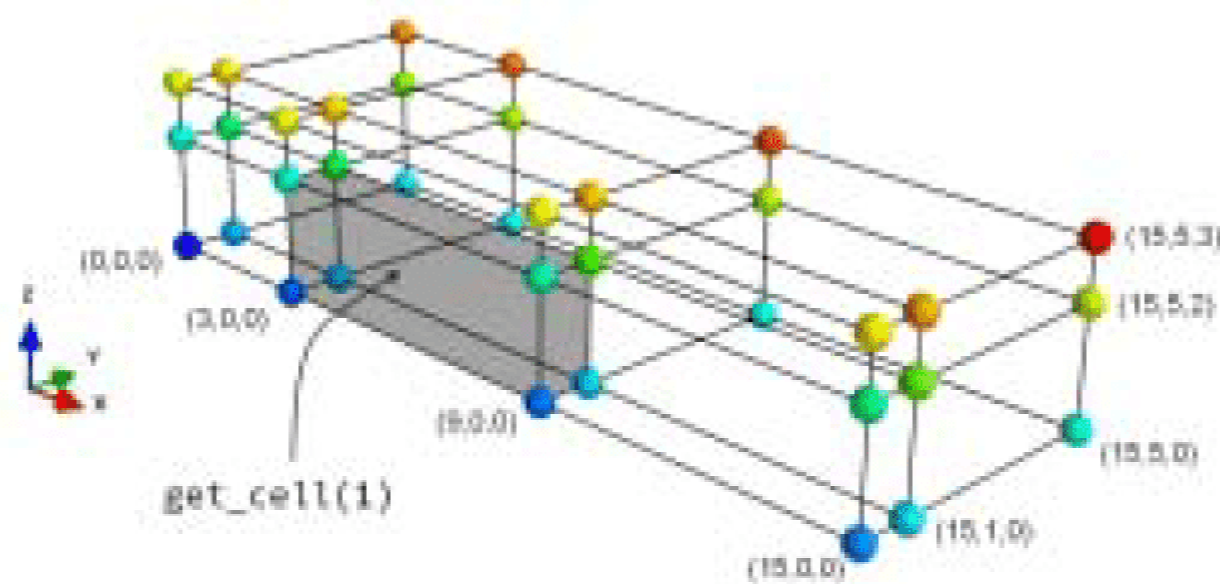


图 9-10　使用 RectilinearGrid 创建分布不均匀的网格

```
x = np.array([0,3,9,15])
y = np.array([0,1,5])
z = np.array([0,2,3])
r = tvtk.RectilinearGrid()
r.x_coordinates = x ❶
r.y_coordinates = y
r.z_coordinates = z
r.dimensions = len(x), len(y), len(z) ❷

r.point_data.scalars = np.arange(0.0,r.number_of_points)
r.point_data.scalars.name = 'scalars'
```

RectilinearGrid 和 ImageData 相同，所有点都在一个正交的网格之上，所不同的是网格的分布是不均匀的。因此需要通过一些属性设置 X 轴、Y 轴和 Z 轴的各个网格平面的位置。❶通过 RectilinearGrid 对象的 x_coordinates、y_coordinates、z_coordinates 属性，可以分别设置网格中与 X 轴、Y 轴和 Z 轴垂直的平面的位置。RectilinearGrid 对象中的点就是所有这些平面的交点。❷由于 RectilinearGrid 对象不会根据这三个数组的长度自动调整 dimensions 属性，因此需要根据数组的长度对 dimensions 属性进行设置。❸最后通过 point_data 属性设置每个点对应的数据。

和 ImageData 对象一样，点的序号依次沿着 X 轴、Y 轴和 Z 轴递增，例如：

```
>>> run tvtk_rectilineargrid.py
>>> for i in xrange(6):
...    print r.get_point(i)
(0.0, 0.0, 0.0)
(3.0, 0.0, 0.0)
(9.0, 0.0, 0.0)
(15.0, 0.0, 0.0)
(0.0, 1.0, 0.0)
(3.0, 1.0, 0.0)
```

单元和点之间的关系也和 ImageData 一样：

```
>>> c = r.get_cell(1)
>>> c.point_ids
[1, 2, 5, 6, 13, 14, 17, 18]
>>> c.points
[(3.0, 0.0, 0.0), (9.0, 0.0, 0.0), (3.0, 1.0, 0.0), (9.0, 1.0, 0.0),
(3.0, 0.0, 2.0), (9.0, 0.0, 2.0), (3.0, 1.0, 2.0), (9.0, 1.0, 2.0)]
```

9.2.3 StructuredGrid

比 RectilinearGrid 更进一步，StructuredGrid 需要我们指定每个点的坐标。而点和单元之间的关系仍然由点在网格中的位置决定。图 9-11 显示了两种用 StructuredGrid 创建的网格结构(见文前彩插)。

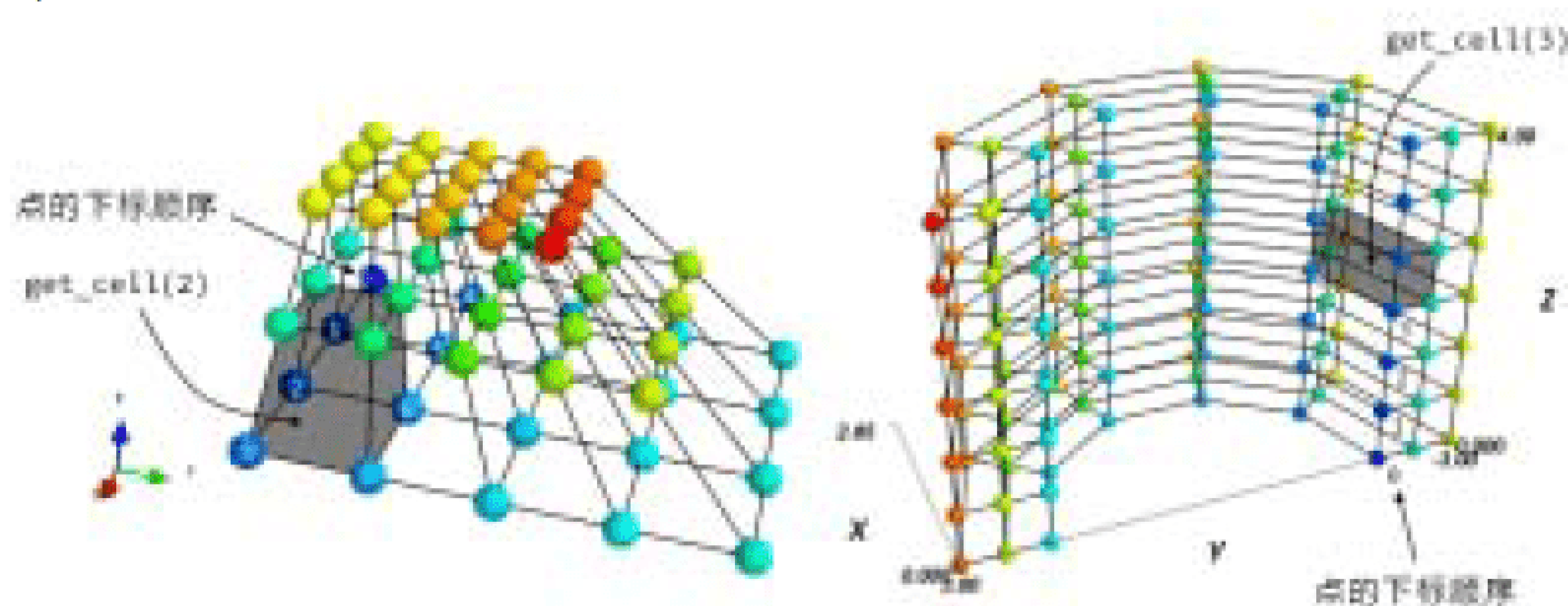


图 9-11 用 StructuredGrid 创建的网格结构

下面对创建这两个网格的程序进行分析。


tvtk_structuredgrid.py
用 StructuredGrid 创建的网格结构

```
def make_points_array(x, y, z):
    return np.c_[x.ravel(), y.ravel(), z.ravel()]

z, y, x = np.mgrid[:3.0, :5.0, :4.0] ❶
x *= (4-z)/3 ❷
y *= (4-z)/3 ❷
s1 = tvtk.StructuredGrid()
s1.points = make_points_array(x, y, z) ❸
s1.dimensions = x.shape[::-1] ❹
s1.point_data.scalars = np.arange(0, s1.number_of_points)
s1.point_data.scalars.name = 'scalars'
```

❶首先用 NumPy 的 mgrid 对象创建了三个数组——x、y 和 z。它们的形状都是(3,5,4)。其中，数组 x 的数据在第 2 轴上变化，数组 y 的数据在第 1 轴上变化，数组 z 的数据在第 0 轴上变化。在这三个数组中，对应下标的数值组成等间距网格上的点。❷将每个点的 X 轴和 Y 轴的坐标值，乘以由 Z 轴坐标值决定的系数。沿着 Z 轴正方向乘积系数逐渐变小，相当于将垂直 Z 轴的网格进行不同比例的收缩。最终形成一个如图 9-11(左)所示的梯形网格。

❸StructuredGrid 对象的 points 属性是数据集中每个点的坐标。它是一个表示 N 个点的坐标的数组，形状为(N, 3)。因此需要将三个形状为(3,5,4)的多维数组合并成一个形状为(3*5*4, 3)的二维数组。这个转换工作由 make_points_array()完成。其中，首先调用参数数组的 ravel()方法得到其平坦化之后的一维数组，然后通过 np.c_对象将三个一维数组按列组合成二维数组。

❹将 StructuredGrid 对象的 dimensions 属性设置为(4,5,3)。在 points 属性中只保存点的坐标，点之间的关系由 dimensions 属性决定。dimensions 和 NumPy 数组的 shape 属性类似，但是其中第 0 轴的变化最快，因此需要将数组的 shape 属性倒序之后再赋值给它。经过 dimensions 属性处理之后，各个点之间的关系如图 9-12 所示。图中每个小方块代表 points 属性中的一个点，方块上的数字表示点在 points 属性中的下标。

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

图 9-12　dimensions 为(4,5,3)的 StructuredGrid 的点的结构

单元由网格中相邻的几个点构成，因此单元 2 由图 9-12 中 8 个灰色矩形表示的点构成：

```
>>> run tvtk_structuredgrid.py
>>> s1.get_cell(2).point_ids
[2, 3, 7, 6, 22, 23, 27, 26]
```

由于单元的形状不是长方体，因此 VTK 采用 Hexahedron 对象表示单元，它有 get_face()和 get_edge()方法分别获得构成此单元的面和边：

```
>>> c = s1.get_cell(2)
>>> type(c)
<class 'tvtk_classes.hexahedron.Hexahedron'>
>>> c.number_of_faces #单元的面数
6
>>> f = c.get_face(0) #获得第 0 个面
>>> type(f) #每个面用一个 Quad 对象表示
<class 'tvtk_classes.quad.Quad'>
>>> f.point_ids #构成第 0 面的 4 个点的下标
[2, 22, 26, 6]
>>> c.number_of_edges # 单元的边数
12
>>> e = c.get_edge(0) #获得第 0 个边
>>> type(e)
<class 'tvtk_classes.line.Line'>
>>> e.point_ids #够成第 0 边的两个点的下标
[2, 3]
```

使用 StructuredGrid 可以创建出任意形状的网格，例如下面的程序可以创建图 9-11(右)所示的半个空心圆柱。程序中，首先在圆柱坐标系中创建等距网格，然后将各点坐标转换到直角坐标系中，具体的步骤留给读者自行分析。

```
r, theta, z2 = np.mgrid[2:3:3j, -np.pi/2:np.pi/2:6j, 0:4:7j]
x2 = np.cos(theta)*r
y2 = np.sin(theta)*r

s2 = tvtk.StructuredGrid(dimensions=x2.shape[::-1])
s2.points = make_points_array(x2, y2, z2)
s2.point_data.scalars = np.arange(0, s2.number_of_points)
s2.point_data.scalars.name = 'scalars'
```

9.2.4 PolyData

PolyData 数据集由一系列的点、点之间的连线以及由点构成的多边形面组成。这些信息都需要用户进行设置，因此用程序创建 PolyData 对象比较繁琐。TVTK 中的许多三维模型类都输出 PolyData 对象。例如在 9.1.1 节中介绍的创建圆锥数据的 ConeSource 类：

```
>>> source = tvtk.ConeSource(resolution = 4)
>>> source.update() # 让 source 计算其输出数据
>>> cone = source.output
```

```
>>> type(cone)
<class 'tvtk_classes.poly_data.PolyData'>
```

我们看到 ConeSource 对象的输出数据是一个 PolyData 对象。PolyData 对象的 points 属性是一个保存点的坐标的数组。为了方便查看各个点的坐标，我们用 NumPy 的 array_str()函数输出此数组的内容，利用 suppress_small 参数将很小的数显示为 0：

```
>>> print np.array_str(cone.points.to_array(), suppress_small=True)
[[ 0.5  0.   0. ]
 [-0.5  0.5  0. ]
 [-0.5 -0.   0.5]
 [-0.5 -0.5 -0. ]
 [-0.5  0.  -0.5]]
```

由于我们设置了 ConeSource 对象的 resolution 属性为 4，因此圆锥的底面是一个正方形。由各个点的坐标很容易看出底面垂直于 X 轴，并且在 X = -0.5 的平面上，而圆锥的顶点在 X 轴上的 X=0.5 处。各个点之间的联系由 polys 属性决定：

```
>>> type(cone.polys)
<class 'tvtk_classes.cell_array.CellArray'>
>>> cone.poly.number_of_cells # 圆锥有 5 个面
5
>>> cone.polys.to_array()
array([4, 4, 3, 2, 1, 3, 0, 1, 2, 3, 0, 2, 3, 3, 0, 3, 4, 3, 0, 4, 1])
```

polys 属性是一个 CellArray 对象，其中保存各个面和点之间的关系。我们创建的圆锥有 5 个面，因此 CellArray 对象的 number_of_cells 属性为 5。CellArray 对象内部使用一个一维整数数组保存构成各个面的点的下标。由于构成每个面的点数可能不同，因此它还需要保存构成每个面的点数。我们可以把这个一维数组理解为如下所示的构造：其中每一行对应一个面，冒号前面的数值是构成此面的点数，冒号后面的一串数字是每个点在 points 属性中的下标。由下面的数据可知，方锥由 4 个三角形和 1 个四边形构成：

```
4 : 4, 3, 2, 1
3 : 0, 1, 2
3 : 0, 2, 3
3 : 0, 3, 4
3 : 0, 4, 1
```

下面我们看看如何直接创建 PolyData 对象。首先是一个简单的方锥的例子：



tvtk_polydata.py
用 PolyData 创建多面体

```
p1 = tvtk.PolyData()
p1.points = [(1,1,0),(1,-1,0),(-1,-1,0),(-1,1,0),(0,0,2)] ❶
faces = [
    4,0,1,2,3,
    3,4,0,1,
    3,4,1,2,
    3,4,2,3,
    3,4,3,0
    ]
cells = tvtk.CellArray() ❷
cells.set_cells(5, faces) ❸
p1.polys = cells
p1.point_data.scalars = np.linspace(0.0, 1.0, len(p1.points))
```

❶在介绍 StructuredGrid 时，我们将一个形状为(N,3)的数组赋值给 points 属性，这里使用坐标列表进行赋值，其效果和使用数组相同。❷为了给 polys 属性赋值，需要首先创建一个新的 CellArray 对象。❸然后调用 CellArray 对象的 set_cells()设置其内容，第一个参数为面(单元)的个数，第二个参数是描述各个面的构成的数组(或列表)。所创建的方锥如图 9-13(左)所示，图中标出了各个点的序号(见文前彩插)。PolyData 对象中的各个面可以通过 get_cell()方法获得，图中标出了第 0 个面和第 1 个面。

```
>>> run tvtk_polydata.py
>>> p1.get_cell(0).point_ids
[0, 1, 2, 3]
>>> p1.get_cell(1).point_ids
[4, 0, 1]
```

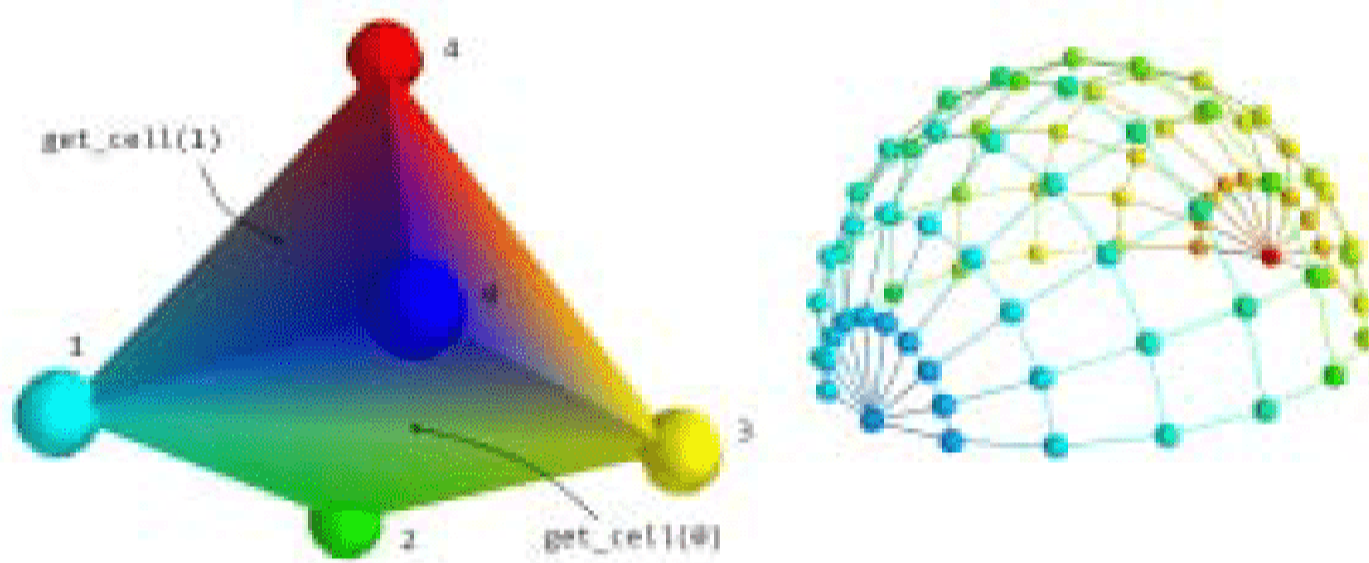


图 9-13　用 PolyData 创建的多面体

下面是使用 PolyData 创建半球面的程序，图 9-13(右)是半球面的显示效果(见文前彩插)。为了显示出整个半球上的点，图中将面隐藏，仅显示各个面的边框：

```
N = 10
a, b = np.mgrid[0:np.pi:N*1j, 0:np.pi:N*1j]
```

```
x = np.sin(a)*np.cos(b)
y = np.sin(a)*np.sin(b)
z = np.cos(a)

from tvtk_structuredgrid import make_points_array
points = make_points_array(x, y, z) ❶
faces = np.zeros(((N-1)**2, 4), np.int) ❷
t1, t2 = np.mgrid[:(N-1)*N:N, :N-1]
faces[:,0] = (t1+t2).ravel()
faces[:,1] = faces[:,0] + 1
faces[:,2] = faces[:,1] + N
faces[:,3] = faces[:,0] + N

p2 = tvtk.PolyData(points = points, polys = faces)
p2.point_data.scalars = np.linspace(0.0, 1.0, len(p2.points))
```


首先将球坐标系中的点转换为直角坐标系中的坐标。❶然后调用前面介绍的 make_points_array()，将这些坐标值转换为形状为(N, 3)的数组。❷如果每个面的点数相同，就可以用一个二维数组表示面和点之间的关系，其中第 0 轴的长度为面数，第 1 轴的长度为每个面的点数。可以将此二维数组直接赋值给 polys 属性，TVTK 库会帮我们完成二维数组到 CellArray 对象的转换：

```
>>> p2.polys.to_array()
array([ 4,  0,  1, 11, 10,  4,  1,  2, 12, 11,  4,  2,  3, 13, 12,  ...])
```

请读者根据程序思考 faces 数组的计算方法，这里就不再多做解释了。

9.3 可视化实例

由于篇幅受限，本书不可能对 VTK 库的用法进行详细的解释。在本节中，我们将对几个比较典型的实例进行分析。希望读者在学习这几个实例之后能够融会贯通，掌握 VTK 开发的一般流程以及使用 TVTK 为开发带来的便利。

从 VTK 的网站可以下载大量的 C++和 TCL 的程序实例，由于 TVTK 的用法更加简洁，读者应该能很容易将它们转换成使用 TVTK 库的 Python 程序。

本节的程序需要使用 VTK 提供的演示数据，如果读者安装了 Python(x,y)，那么这些演示数据可以在“C:\Python26\VTKData”中找到。此路径已经在环境变量 VTK_DATA_ROOT 中定义，读者可以输入下面的语句验证一下：

```
>>> import os
>>> os.environ["VTK_DATA_ROOT"]
'C:\\Python26\\VTKData'
```

如果读者的计算机上没有演示数据和环境变量，请将本书附带光盘中的 VTKData 文件夹复制到本章的源程序文件夹中。

在本节的可视化实例程序中，将使用下面的辅助模块 utility。其中的 vtk_data()获得 VTK 数据文件的路径，而 show_actors()则使用 ivtk 显示一组 Actor 对象。



utility.py
可视化实例的辅助函数

```
import os
import os.path as path
from enthought.tvtk.tools import ivtk
from enthought.pyface.api import GUI

def vtk_data(name):
    folder = os.environ.get("VTK_DATA_ROOT", "VTKData")
    datapath = os.path.join(folder, "data", name)
    if not path.exists(datapath):
        raise IOError("please set environment variable: VTK_DATA_ROOT")
    return datapath

def show_actors(actors, shell=False):
    gui = GUI()
    if shell:
        window = ivtk.IVTKWithCrustAndBrowser(size=(800,600))
    else:
        window = ivtk.IVTKWithBrowser(size=(600,400))
    window.open()
    for a in actors:
        window.scene.add_actor( a )
    window.scene.background = (1,1,1)
    return window, gui
```

9.3.1 切面

展示三维数据的一个比较简单的方法是使用切面，将切面经过的数据进行可视化，这样就把一个三维数据的可视化问题转换成了二维数据的可视化。通过交互式地修改切面的位置和方向，用户能直观地对三维数据进行观察。下面是使用切片工具观察数据的一个实例，效果如图 9-14 所示。它由三部分组成：一个曲面、一个平面和一个外框。在曲面和平面上，每个点通过颜色表示其对应的数值的大小。由于我们使用 ivtk 显示可视化结果，因此可以使用界面左侧的流水线浏览器观察组成整个场景的流水线。



tvtk_cut_plane.py
使用切面观察三维数据

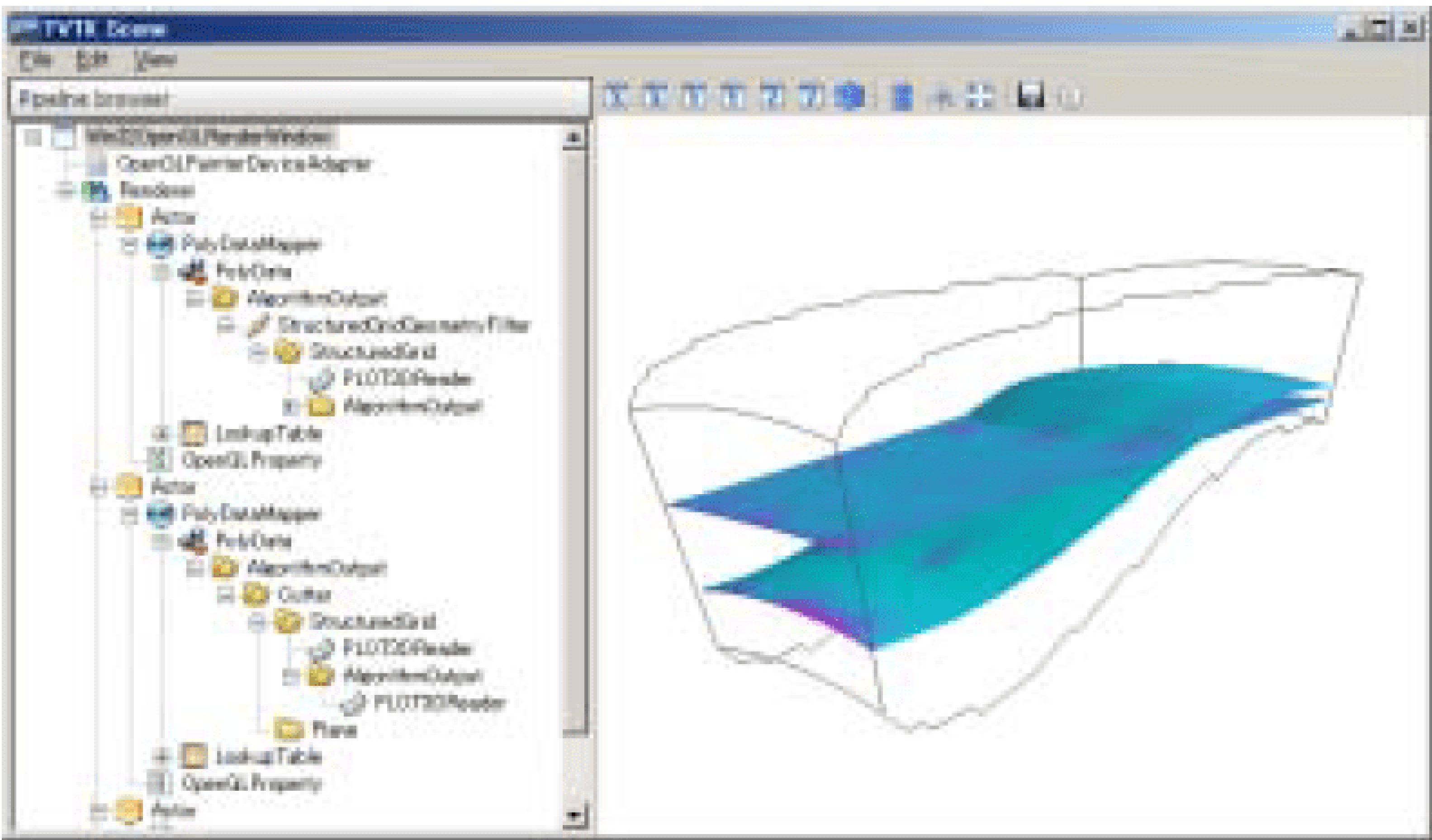


图 9-14　使用切面观察 StructuredGrid 数据集

```
import numpy as np
from enthought.tvtk.api import tvtk
from utility import vtk_data, show_actors

def read_data():
    # 读入数据
    plot3d = tvtk.PLOT3DReader( ❶
        xyz_file_name = vtk_data("combxyz.bin"),
        q_file_name = vtk_data("combq.bin"),
        scalar_function_number = 100, vector_function_number = 200
    )
    plot3d.update() ❷
    return plot3d

if __name__ == "__main__":
    plot3d = read_data()

    # 创建颜色映射表
    lut = tvtk.LookupTable() ❸
    import pylab as pl
    lut.table = pl.cm.cool(np.arange(0,256))*255  ❹

    # 显示 StructuredGrid 中的一个网格面
    plane = tvtk.StructuredGridGeometryFilter( ❺
        input = plot3d.output, extent = (0, 100, 0, 100, 6, 6)
    )
    plane_mapper = tvtk.PolyDataMapper(lookup_table = lut, input = plane.output) ❻
    plane_mapper.scalar_range = plot3d.output.scalar_range ❼
    plane_actor = tvtk.Actor(mapper = plane_mapper) ❽
```

下面先看看曲面的制作过程。❶为了对可视化方法进行重点介绍，我们直接使用一个PLOT3DReader 对象从数据文件读入三维数据信息。❷为了让 PLOT3DReader 对象真正从文件读入数据，需要调用它的 update()方法。运行此方法之后，就可以查看 output 属性所表示的数据集了。

```
>>> run tvtk_cut_plane.py
>>> type(plot3d.output)
<class 'tvtk_classes.structured_grid.StructuredGrid'>
```

PLOT3DReader 读入 PLOT3D 格式的文件，PLOT3D 是一个用于对流体力学数据进行可视化的程序。程序中通过设置 scalar_function_number 和 vector_function_number 属性，分别指定标量数组和矢量数组为流体的密度和速度。各个参数的具体含义请读者查看 VTK 的相关文档。PLOT3DReader 对象输出的是一个 StructuredGrid 数据集。下面观察它的一些属性：

```
>>> s = plot3d.output
>>> s.dimensions
array([57, 33, 25])
>>> s.points
[(2.6670000553131104, -3.7747600078582764, 23.832920074462891), ...,
 (16.510000228881836, 5.6621408462524414, 35.749378204345703)], length = 47025
>>> s.cell_data.number_of_arrays
0
>>> s.point_data.number_of_arrays
4
>>> for i in xrange(4):
...     print s.point_data.get_array_name(i)
Density
Momentum
StagnationEnergy
Velocity
>>> s.point_data.scalars.name
Density
>>> s.point_data.vectors.name
Velocity
```

由上面的各个属性可以得知，plot3d 对象输出的数据集是一个形状为(57, 33, 25)的网格。由于它是一个 StructuredGrid 对象，因此网格中每个点的坐标保存在 points 属性中。网格中的单元没有数据，而每个点对应 4 种数据，它们的名字分别为“Density”、“Momentum”、“StagnationEnergy”和“Velocity”。其中，通过 point_data 的 scalars 属性可以获得名为“Density”的数组，它是一个标量数组，而“Velocity”数组则可以通过 point_data 的 vectors 属性获得，它是一个三维矢量数组。切面将针对 scalars 属性的数据进行运算，因此切面所显示的是切面上每个点的密度值。

❸我们需要把切面上的密度用某种颜色来表示，因此需要进行值到颜色的转换。在 VTK 中，这种转换工作由 LookupTable 对象完成。❹LookupTable 对象的 table 属性是一个保存颜色表的数组。这里使用 matplotlib 库的颜色映射对象来计算 LookupTable 对象的颜色。

LookupTable 对象的 range 属性保存值的范围，它将这个范围之内的值映射到 table 属性的颜色表之上。在本例中，LookupTable 对象的取值范围和密度数组的范围一致，而密度数组的范围可以通过 StructuredGrid 对象的 scalar_range 属性快速获得：

```
>>> lut.range
array([ 0.19781309,  0.71041924])
>>> s.point_data.scalars.range
(0.19781309366226196, 0.71041923761367798)
>>> s.scalar_range
(0.19781309366226196, 0.71041923761367798)
```

❺StructuredGridGeometryFilter 对象能从 StructuredGrid 对象中提取部分网格。程序中通过 extent 属性指定了提取的网格的范围：第 0 轴和第 1 轴的范围是 0 到 100，而第 2 轴的范围是 6 到 6。其中，第 0、1 轴的范围大于网格在这两个方向上的长度，因此它们提取这两个轴上的整个范围，而在第 2 轴上只提取第 6 层的数据。StructuredGridGeometryFilter 对象输出一个 PolyData 数据集：

```
>>> p = plane.output
>>> type(p)
<class 'tvtk_classes.poly_data.PolyData'>
```

下面查看这个 PolyData 对象的一些属性：

```
>>> p.number_of_points
1881
>>> s.dimensions[0]*s.dimensions[1]
1881
```

它由 1881 个点构成，因为它是 StructuredGrid 对象中第 2 轴上的一层，所以它的点数等于原始数据集的第 0 轴长度和第 1 轴长度的乘积。下面比较 StructuredGrid 对象和 PolyData 对象中点的坐标：

```
>>> s.dimensions
array([57, 33, 25])
>>> tmp1 = s.points.to_array().reshape((25,33,57,3))
>>> tmp2 = p.points.to_array().reshape((33,57,3))
>>> np.all(tmp1[6] == tmp2)
True
```

tmp1 是将 StructuredGrid 对象的点还原成三维网格之后的数组，数组的第 3 轴表示每个点的X轴、Y轴和Z轴的坐标值。由于NumPy数组的shape属性和StructuredGrid对象的dimensions属性之间是倒序关系，因此 dimensions 属性的第 0 轴相当于 shape 属性的第 2 轴。同样，我们将 PolyData 对象的点还原成一个表示二维网格的数组 tmp2。由于我们提取的是 StructuredGrid 对象中第 2 轴的第 6 层数据，因此 tmp1[6]应该和 tmp2 相等。

使用 StructuredGridGeometryFilter 对象不但从数据源提取了点的坐标，而且还提取了每个点所对应的数据，因此 PolyData 对象的每个点也对应有 4 组数据，并且数组名保持不变：

```
>>> p.point_data.number_of_arrays
4
>>> p.point_data.scalars.name
'Density'
```

❻使用 PolyDataMapper 对象将 PolyData 对象转换为图形数据，同时用前面创建的颜色映射表 lut 对 PolyData 对象中的每个点进行着色。❼为了能使用颜色映射表中的所有颜色，我们将颜色映射表的范围设置成和密度数组的范围一致。❽最后创建在场景中显示图形数据的 Actor 对象。

接下来分析创建平面切面的程序：

```
# 做一个平面切面
cut_plane = tvtk.Plane(origin = plot3d.output.center, normal=(-0.287, 0, 0.9579)) ❶
cut = tvtk.Cutter(input = plot3d.output, cut_function = cut_plane) ❷
cut_mapper = tvtk.PolyDataMapper(input = cut.output, lookup_table = lut)
cut_actor = tvtk.Actor(mapper = cut_mapper)
```

❶首先创建表示平面的 Plane 对象。它是一个经过 origin 属性表示的坐标点，法线方向为 normal 属性的无限平面，通过这两个属性可以唯一确定平面。Plane 对象的输出是一个 PolyData 对象：

```
>>> type(plane.output)
<class 'tvtk_classes.poly_data.PolyData'>
```

❷创建一个 Cutter 对象，它使用传递给 cut_function 属性的 Plane 对象，对 input 属性的 StructuredGrid 对象进行切面计算。Cutter 对象的输出也是 PolyData 对象，由于 StructuredGrid 对象中的点不一定能正好落在Plane对象所指定的平面之上，因此Cutter对象会对StructuredGrid 对象中的数据进行插值计算。它所输出的 PolyData 对象中的点不再是数据源中的点。

```
>>> cut.output.number_of_points
2537
```

但是，PolyData 对象中的每个点仍然与 4 组数据相对应，这些数据都是通过对数据源的数据进行插值计算得到的：

```
>>> cut.output.point_data.number_of_arrays
4
```

为了在场景中显示切面，就像前面的曲面一样，可以使用 PolyDataMapper 和 Actor 将 PolyData 对象加工成场景中的物体。

接下来创建 StructuredGrid 对象的外边框部分：

```
# StructuredGrid 网格的外边框
outline = tvtk.StructuredGridOutlineFilter(input = plot3d.output)
outline_mapper = tvtk.PolyDataMapper(input = outline.output)
outline_actor = tvtk.Actor(mapper = outline_mapper)
outline_actor.property.color = 0.3, 0.3, 0.3
```

可以使用 StructuredGridOutlineFilter 对象计算出一个表示外边框的 PolyData 对象。请读者根据前面介绍的方法观察此 PolyData 对象的内容，这里就不再举例了。

最后使用 utility 模块中定义的 show_actors()显示前面创建的 3 个 Actor 对象——plane_actor、cut_actor 和 outline_actor：

```
win, gui = show_actors([plane_actor, cut_actor, outline_actor])
gui.start_event_loop()
```

在界面左侧的流水线浏览器中，双击某个对象可以打开它对应的编辑器，对对象的各种属性进行编辑。例如，可以打开 Plane 对象的编辑器。在此编辑器中修改 Plane 对象的 original 和 normal 属性，从而改变切面的位置和方向，观察数据集中不同位置的密度分布情况。打开 StructuredGridGeometryFilter 对象的编辑器，可以修改曲面切面的范围。图 9-15 显示了这两个编辑器，以及通过它们修改之后的切面(见文前彩插)。

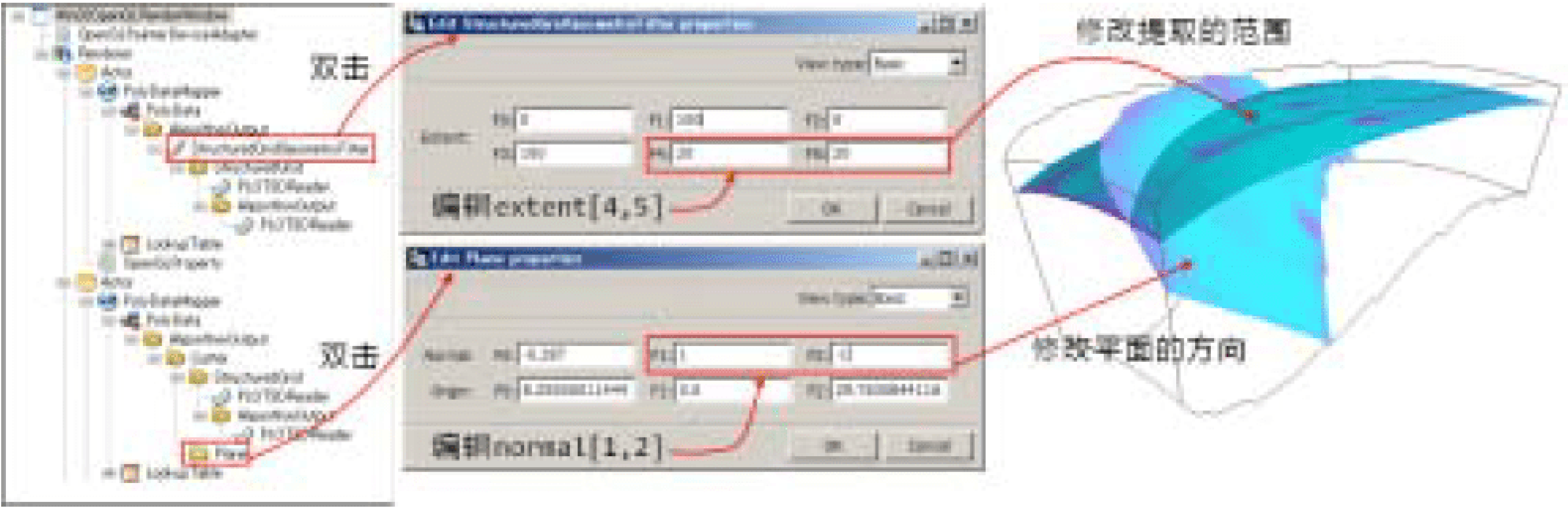


图 9-15　通过编辑器修改切面的位置和方向

9.3.2　等值面

等值面是标量场中标量值相等的曲面，和地图中的等高线类似。在 VTK 中，使用 Contour-

Filter 计算等值面。下面的程序可以对前面的流体数据进行等值面可视化，效果如图 9-16 所示(见文前彩插)。



tvtk_contours.py
使用等值面对标量场进行可视化

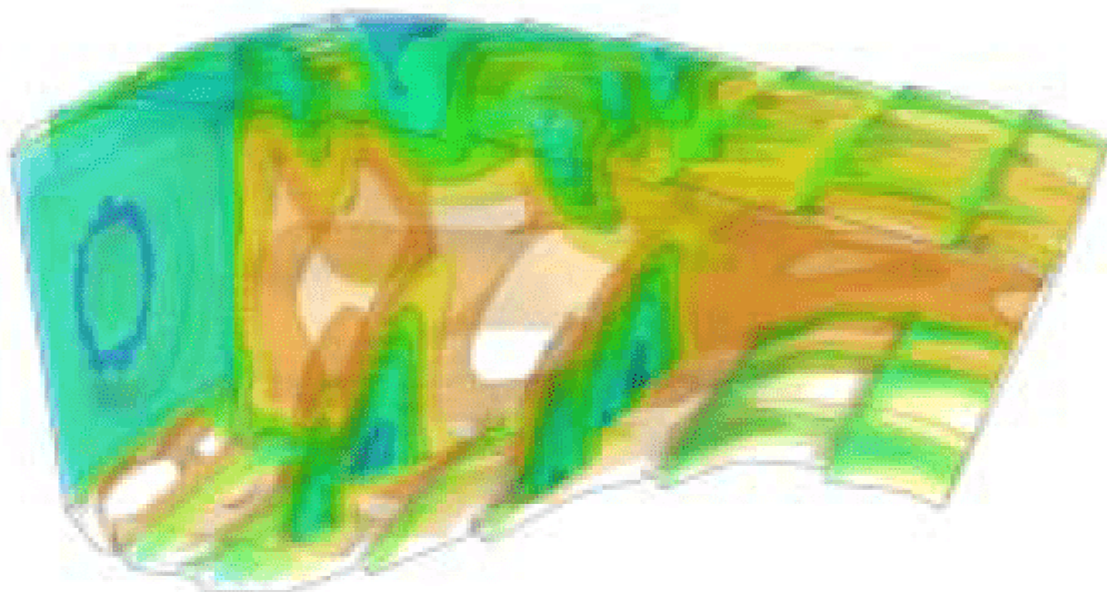


图 9-16　使用等值面对标量场进行可视化

```
plot3d = read_data()
contours = tvtk.ContourFilter(input = plot3d.output)
contours.generate_values(8, plot3d.output.point_data.scalars.range) ❶
mapper = tvtk.PolyDataMapper(input = contours.output,
    scalar_range = plot3d.output.point_data.scalars.range) ❷
actor = tvtk.Actor(mapper = mapper)
actor.property.opacity = 0.3 ❸
```

首先使用前面介绍的 read_data()函数读入数据。❶创建一个 ContourFilter 对象，并且调用其 generate_values()方法创建 8 个等值面，等值面的取值范围由标量值数组 scalars 的范围决定。❷等值面的颜色映射也由 scalars 数组的范围决定。由于没有设置映射器的颜色表，因而将使用系统默认的映射表，将最小值映射为红色，将最大值映射为蓝色。

由于 8 个等值面是嵌套的，我们需要修改等值面的 Actor 对象的透明度，以便观察等值面的内部构造。除了使用 generate_values()创建等差等值面之外，还可以使用 set_value()方法直接设置每个等值面的值，而使用 get_value()方法则可以获得等值面的值。下面的代码在 IPython 中修改第 0 个等值面的值：

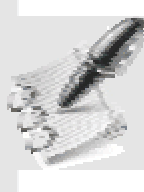
```
>>> run tvtk_contours.py
>>> contours.get_value(0)
0.19781309366226196
>>> contours.set_value(0, 0.21)
```

修改之后，只需要旋转或移动一下场景就能看到所作的修改。也可以使用下面的代码强制

场景重新绘制：

```
>>> win.scene.render()
```

在这个例子中，同一个等值面上所有点的颜色是相同的。因为等值面上的标量值(流体密度)相同，而对等值面进行着色时，默认也使用标量值。但有时候，我们希望等值面的颜色由另外的标量值决定。下面的程序演示了如何使用别的标量值对等值面进行着色。



tvtk_contours2.py
使用别的标量对等值面进行着色

```
plot3d = read_data()
plot3d.add_function(153) ❶
plot3d.update()
contours = tvtk.ContourFilter(input = plot3d.output)
contours.set_value(0, 0.32) ❷
mapper = tvtk.PolyDataMapper(input = contours.output,
    scalar_range = plot3d.output.point_data.get_array(4).range, ❸
    scalar_mode = "use_point_field_data") ❹
mapper.color_by_array_component("VelocityMagnitude", 0) ❺
actor = tvtk.Actor(mapper = mapper)
actor.property.opacity = 0.6
```

❶首先调用 PLOT3DReader 对象的 add_function()方法，为每个点添加一组标量值。所增加的新数组名为 VelocityMagnitude，表示每个点对应的速度大小，我们将使用此数组对等值面进行着色：

```
>>> run tvtk_contours2.py
>>> plot3d.output.point_data.get_array_name(4)
'VelocityMagnitude'
```

❷调用 ContourFilter 对象的 set_value()方法，创建一个值为 0.32 的等值面。❸设置映射器的标量范围属性 scalar_range，将它设置为新增加的数组的取值范围。❹scalar_mode 属性决定映射器所使用的标量数据类型，这里"use_point_field_data"表示使用点数据中的数组，它有如下几种选择：

- "default"：使用 point_data.scalars，如果不存在就使用 cell_data.scalars。
- "use_point_data"：使用 point_data.scalars。
- "use_cell_data"：使用 cell_data.scalars。
- "use_point_field_data"：使用 point_data 中的某个数组，具体的数组需要另外指定。
- "use_cell_field_data"：使用 cell_data 中的某个数组，具体的数组需要另外指定。

❺通过 color_by_array_component()指定着色所使用的数据。它的第一个参数是数组名，第二个参数是从数组中所选择的列。由于"VelocityMagnitude"是一个标量数组，它只有一列数据，因此第二个参数为 0。如果第一个参数是矢量数组名，例如"Velocity"，那么可以通过第二个参数选择矢量数组中的不同分量。例如，0 表示选择 X 轴方向的速度，1 表示选择 Y 轴方向的速度，2 表示选择 Z 轴方向的速度。请读者修改这两个参数，使用其他的数据对等值面进行着色。

9.3.3 流线

在前面的实例中，我们使用切面和等值面对流体的密度分布进行了可视化。空间中每一点的密度都可以用一个数值(标量)来表示，因此我们将密度分布理解为一个标量场。而流体在每一点的速度是一个矢量，因此速度的分布情况需要使用矢量场来描述。本节介绍如何使用随机散布的矢量箭头和流线对矢量场进行可视化，效果如图 9-17 所示(见文前彩插)。由此图可知，整个场景由 4 个实体构成：外框、随机散布的矢量箭头、表示流线源的球体以及流线。

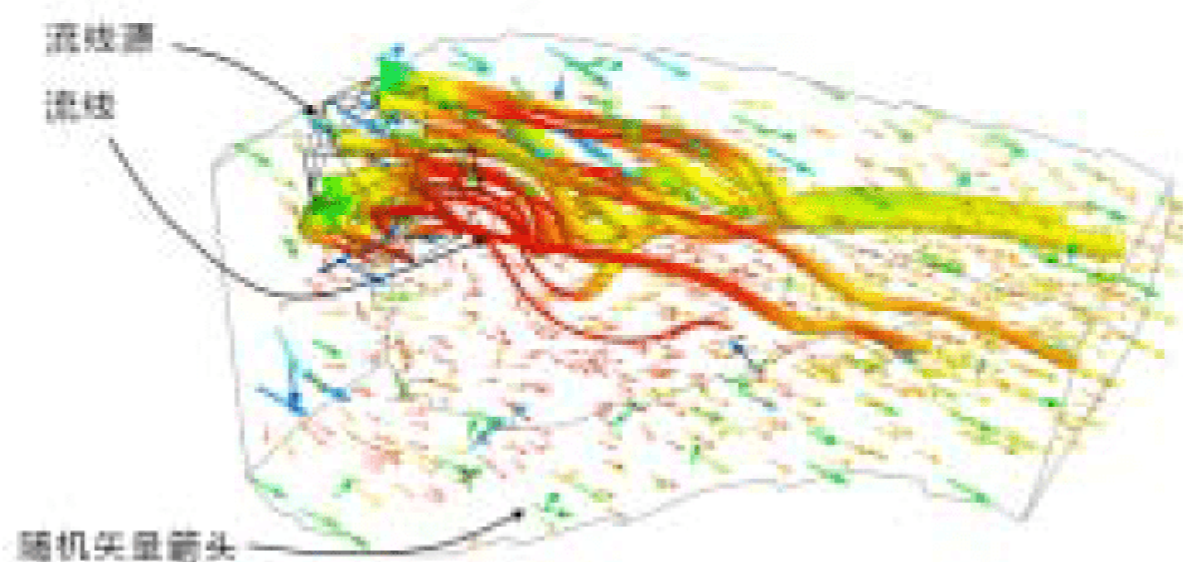


图 9-17 矢量场的可视化



tvtk_streamline.py
使用流线和随机散布的矢量箭头可视化矢量场

```
# 读入数据
plot3d = read_data()

# 矢量箭头
mask = tvtk.MaskPoints(input = plot3d.output, random_mode=True, on_ratio=50) ❶
arrow_source = tvtk.ArrowSource() ❷
arrows = tvtk.Glyph3D(input = mask.output, source=arrow_source.output, ❸
    scale_factor=2/np.max(plot3d.output.point_data.scalars.to_array()))
arrows_mapper = tvtk.PolyDataMapper(input = arrows.output,
    scalar_range = plot3d.output.point_data.scalars.range)
arrows_actor = tvtk.Actor(mapper = arrows_mapper)
```

上面的代码创建随机散布的矢量箭头。这些箭头的起始点是数据集中点的坐标，箭头的方

向由点对应的矢量决定，而箭头的大小和颜色则由点对应的标量决定。本例中，箭头的方向表示了速度的方向，而大小和颜色则表示了密度。箭头越大表示该点的标量值(密度)越大，箭头的颜色也同时表示标量值的大小，红色对应的标量值最小，而蓝色对应的标量值最大。

❶由于原始数据集中的点数很多，如果在所有点的位置都描绘箭头，将十分耗时并且无法区分众多的箭头。因此首先使用 MaskPoints 对象对数据集中的数据进行随机选取，某点被选中的概率为 1/50 ，即每 50 个点选择一个点。下面的代码查看 MaskPoints 对象输出的数据类型和点数，以及每个点所对应的数据：

```
>>> run tvtk_streamline.py
>>> plot3d.output.number_of_points
47025
>>> type(mask.output)
<class 'tvtk_classes.poly_data.PolyData'>
>>> mask.output.number_of_points
930
>>> mask.output.point_data.number_of_arrays
4
```

❷ArrowSource 对象创建表示箭头的 PolyData 数据集。❸Glyph3D 对象对箭头数据进行复制，在 masks 输出的 PolyData 数据集的每个点上都放置一个箭头。箭头的方向、长度和颜色由与点对应的矢量和标量数据决定。scale_factor 参数是所有箭头的共同缩放系数，这里使用标量数组的最大值对缩放系数进行正规化。Glyph3D 对象可以对任意的 PolyData 数据进行复制，读者可以尝试将箭头数据源 ArrowSource 改为圆锥数据源 ConeSource：

```
>>> arrow_source.output.number_of_points # 一个箭头有 31 个点
31
>>> arrows.output.number_of_points # 箭头被复制了 930 份，因此有 930*31 个点
28830
```

我们还可以使用流线直观地观察矢量场。流线上每一点的切线方向就是矢量场在该点的方向。下面是显示流线的代码：

```
# 作为流线起点的球
center = plot3d.output.center
sphere = tvtk.SphereSource( ❶
    center=(2, center[1], center[2]), radius=2,
    phi_resolution=6, theta_resolution=6)
sphere_mapper = tvtk.PolyDataMapper(input = sphere.output)
sphere_actor = tvtk.Actor(mapper = sphere_mapper)
sphere_actor.property.set(
    representation = "wireframe", color=(0,0,0))
```

```
# 流线
streamer = tvtk.StreamLine( ❷
    input = plot3d.output,
    source = sphere.output,
    step_length = 0.0001,
    integration_direction = "forward",
    integrator = tvtk.RungeKutta4()) ❸

tube = tvtk.TubeFilter( ❹
    input = streamer.output,
    radius = 0.05,
    number_of_sides = 6,
    vary_radius = "vary_radius_by_scalar")

tube_mapper = tvtk.PolyDataMapper(
    input = tube.output,
    scalar_range = plot3d.output.point_data.scalars.range)
tube_actor = tvtk.Actor(mapper = tube_mapper)
tube_actor.property.backface_culling = True
```

❶SphereSource 对象创建表示球体的 PolyData 数据集。通过 center 和 radius 属性指定球体的球心位置和半径。phi_resolution 和 theta_resolution 属性指定球体的经度和纬度方向上的等分次数，分割得越细，输出的 PolyData 数据集越接近球体。

❷StreamLine 对象在矢量场中计算流线。计算流线的起点坐标由其 source 属性中点的坐标决定。这里我们以 SphereSource 对象输出的球体上的点作为流线的计算起点。如果通过 start_position 属性设置起始点的坐标，将只计算一条流线。step_length 属性决定了流线上点的间隔，此属性值越小，流线上的点越多，流线越平滑，计算所需的时间也越长。integration_direction 属性决定流线的计算方向，值可以为'backward'、'forward'或'integrate_both_directions'。'forward' 表示计算起点之后的流线，'backward'表示计算起点之前的流线。❸流线的计算需要使用由 integrator 属性指定的数值积分算法，这里使用 RungeKutta4，它是一个 4 阶 Runge-Kutta 积分算法。

StreamLine 对象的输出是一个 PolyData 数据集，它的所有单元都是表示流线的路径，因此边(line)数为 23，面数为 0：

```
>>> streamer.output.number_of_points
5529
>>> streamer.output.number_of_polys
0
>>> streamer.output.number_of_lines
23
```

❹使用 TubeFilter 对象可以将流线路径转换为有粗细的圆管。radius 属性为圆管的粗细，number_of_side 属性指定圆管的切面圆的边数。圆管的粗细可以根据点的数据发生变化，这里使用"vary_radius_by_scalar"指定圆管的粗细由标量(密度)决定。

TubeFilter 对象的输出也是 PolyData 数据集，它由众多的面构成，但是它的 number_of_polys 属性却等于 0：

```
>>> tube.output.number_of_polys
0
```

这里为了节省内存空间和计算时间，PolyData 对象使用 TriangleStrip 对象表示三角面。一个 TriangleStrip 对象由一组点构成。每连续的三个点都将构成一个三角面：

```
>>> tube.output.number_of_strips
138
>>> t = tube.output.get_cell(0)
>>> type(t)
<class 'tvtk_classes.triangle_strip.TriangleStrip'>
>>> t.number_of_points
498
```

在上面的例子中，箭头的大小和颜色、流线的粗细和颜色所表示的是流体的密度。有时候，我们希望用这些可视化元素表示矢量的长度，即流体的速度大小。可以直接计算 vectors 数组中各个矢量的长度，并且将结果写入数据集的 scalars 数组中。例如，在读入数据之后如下添加两行代码：

```
# 读入数据
plot3d = read_data()
## 添加下面两行代码
p = plot3d.output.point_data
p.scalars = np.sqrt(np.sum(p.vectors.to_array()**2, axis=-1))
```

或者使用 TVTK 库中的 VectorNorm，它可以计算输入数据的 vectors 数组中各个矢量的长度，并且将结果保存到 scalars 数组中：

```
# 读入数据
plot3d = read_data()
## 添加下面两行代码
plot3d = tvtk.VectorNorm(input=plot3d.output)
plot3d.update()
```

9.4 TVTK 的改进

作为本章的最后一节，我们总结一下 TVTK 库对 VTK 库的改进。首先看一个使用标准的 VTK 库 API 显示圆锥的例子：



vtk_cone_example.py
调用标准的 VTK 库 API 绘制圆锥

```
import vtk

# 创建一个圆锥数据源
cone = vtk.vtkConeSource( )
cone.SetHeight( 3.0 )
cone.SetRadius( 1.0 )
cone.SetResolution(10)
# 使用 PolyDataMapper 将数据转换为图形数据
coneMapper = vtk.vtkPolyDataMapper( )
coneMapper.SetInput( cone.GetOutput( ) )
# 创建一个 Actor
coneActor = vtk.vtkActor( )
coneActor.SetMapper ( coneMapper )
# 用线框模式显示圆锥
coneActor.GetProperty( ).SetRepresentationToWireframe( )
# 创建 Renderer 和窗口
ren1 = vtk.vtkRenderer( )
ren1.AddActor( coneActor )
ren1.SetBackground( 0.1 , 0.2 , 0.4 )
renWin = vtk.vtkRenderWindow( )
renWin.AddRenderer( ren1 )
renWin.SetSize(300 , 300)
# 创建交互工具
iren = vtk.vtkRenderWindowInteractor( )
iren.SetRenderWindow( renWin )
iren.Initialize( )
iren.Start( )
```

此程序和 C++程序的区别仅仅是没有声明变量的类型，其他的用法完全和 C++的 VTK API 相同。官方提供的 VTK-Python 包和 C++语言的接口相似，许多地方没有体现出 Python 作为动态语言的优势，可以说标准的 VTK-Python 库不是 Python 风格的。为了弥补这些不足之处，Enthought 公司开发的 TVTK 库进一步对 VTK-Python 进行封装，它具有如下优点：

- 支持 Trait 属性
- 支持元素的 Pickle 操作
- API 更接近 Python 风格
- 能自动处理 NumPy 数组或列表对象
- 流水线浏览器 ivtk

9.4.1 TVTK 的基本用法

下面是前面介绍过的用 TVTK 显示圆锥的程序：

```
from enthought.tvtk.api import tvtk

cs = tvtk.ConeSource(height=3.0, radius=1.0, resolution=36)
m = tvtk.PolyDataMapper(input = cs.output)
a = tvtk.Actor(mapper=m)
ren = tvtk.Renderer(background=(1, 1, 1))
ren.add_actor(a)
rw = tvtk.RenderWindow(size=(300,300))
rw.add_renderer(ren)
rwi = tvtk.RenderWindowInteractor(render_window=rw)
rwi.initialize()
rwi.start()
```

这个程序比标准 VTK 版本的要简短许多，从中我们可以看出 TVTK 的一些重要改进：

- TVTK 库中的类名除去了前缀"vtk"。有些类名在"vtk"之后是数字，TVTK 库将对这种类名进行特殊处理：如果首字符为数字，就用其英文单词代替。例如，vtk3DSImporter 变成 ThreeDSImporter。
- 函数名按照 Enthought 库的惯例，采用下划线来连接单词，例如 AddItem 将变成 add_item。
- 许多 VTK 对象的方法在 TVTK 中用 Trait 属性替代，例如下面代码列出了圆锥实例中的两处改进：

```
m.SetInput(cs.GetOutput())       # VTK
m.input = cs.output             # TVTK

p.SetRepresentationToWireframe()  # VTK
p.representation = 'w'           # TVTK
```

- Trait 属性可以在创建对象的同时通过关键字参数进行设置，这样更便于程序的编写和阅读。

在 TVTK 库的内部实现中，所有 TVTK 对象的内部都有一个 VTK 对象，对 TVTK 对象的函数调用将转给内部的 VTK 对象来执行。如果返回值是 VTK 对象，它将被封装成 TVTK 对

象返回。如果方法的参数是 TVTK 对象，其中的 VTK 对象将作为值进行参数传递。

9.4.2 Trait 属性

所有的 TVTK 类都从 HasStrictTraits 继承，HasStrictTraits 规定其子类的对象在创建之后不能对不存在的属性进行赋值。

VTK 中所有和基本状态有关的方法在 TVTK 中都使用 Trait 属性表示。使用 Trait 属性有如下优点：

- 调用 set()方法可以一次设置多个 Trait 属性，例如：

```
>>> p = tvtk.Property()
>>> p.set(opacity=0.5, color=(1,0,0), representation="w")
```

- 调用 edit_traits()或 configure_traits()可以显示编辑属性的对话框：

```
>>> p.edit_traits()
```

也可以通过 tvtk.to_tvtk()得到任何 TVTK 对象所封装的 TVK 对象，在必要的时候直接对 VTK 对象进行操作：

```
>>> print p.representation
wireframe
>>> p_vtk = tvtk.to_vtk(p)
>>> p_vtk.SetRepresentationToSurface()
>>> print p.representation
surface
```

还可以通过 TVTK 对象的_vtk_obj 属性获得其中的 VTK 对象，通过 tvtk.to_tvtk(p)则可以将 VTK 对象 p 封装成 TVTK 对象。

9.4.3 序列化(Pickling)

TVTK 对象支持简单的序列化处理。单个 TVTK 对象的状态可以被序列化：

```
>>> import cPickle
>>> p = tvtk.Property()
>>> p.representation="w"
>>> s = cPickle.dumps(p)
>>> del p
>>> q = cPickle.loads(s)
>>> q.representation
'wireframe'
```

但是序列化仅仅能保存对象的状态，对象之间的引用无法被保存，因此 TVTK 的整个流水

线无法用序列化保存。

通常 pickle.load()可创建新的对象，如果希望更新某个已经存在的对象的状态，可以通过调用__setstate__()来实现：

```
>>> p = tvtk.Property()
>>> p.interpolation = "flat"
>>> d = p.__getstate__()
>>> del p
>>> q = tvtk.Property()
>>> q.interpolation
'gouraud'
>>> q.__setstate__(d)
>>> q.interpolation
'flat'
```


9.4.4 集合迭代

从 tvtk.Collection 继承的对象可以像标准的 Python 序列对象一样使用，下面的例子演示了 ActorCollection 对象能够支持 len()、append()以及 for 循环：

```
>>> ac = tvtk.ActorCollection()
>>> len(ac)
0
>>> ac.append(tvtk.Actor())
>>> ac.append(tvtk.Actor())
>>> len(ac)
2
>>> for a in ac:
...     print a
...
vtkOpenGLActor (06A99EB8)
......
vtkOpenGLActor (069C4270)
......
>>> del ac[0]
>>> len(ac)
1
```

对比一下 VTK 库相应的版本，就能体会出 TVTK 库的优点了：

```
>>> ac = vtk.vtkActorCollection()
>>> ac.GetNumberOfItems()
0
```

```
>>> ac.AddItem(vtk.vtkActor())
>>> ac.AddItem(vtk.vtkActor())
>>> ac.GetNumberOfItems()
2
>>> ac.InitTraversal()
>>> for i in range(ac.GetNumberOfItems()):
...     print ac.GetNextItem()
...
vtkOpenGLActor (05E0A750)
......
vtkOpenGLActor (05E0A8C0)
......
>>> ac.RemoveItem(0)
>>> ac.GetNumberOfItems()
1
```

9.4.5　数组操作

DataArray 的所有派生类和 Python 的序列一样，支持迭代接口，以及__getitem__()、__setitem__()、__repr__()、append()、extend()等方法。此外，它还可以通过 from_array()直接用 NumPy 数组或列表进行赋值。可以很方便地将其中保存的数据转换为 NumPy 数组。Points 和 IdList 对象也同样支持这些特性：

```
>>> pts = tvtk.Points()
>>> p_array = np.eye(3)
>>> p_array
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> pts.from_array(p_array)
>>> pts.print_traits()
_in_set:                 0
_vtk_obj:                <vtkCommonPython.vtkPoints vtkobject at 069A0FB0>
actual_memory_size:       1L
bounds:                  (0.0, 1.0, 0.0, 1.0, 0.0, 1.0)
class_name:               'vtkPoints'
data:                    [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]
data_type:                'double'
...
number_of_points:         3
reference_count:          1
>>> pts.to_array()
array([[ 1.,  0.,  0.],
```

```
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

如果 TVTK 对象的属性或方法能够接受 DataArray、Points、IdList、CellArray 等对象，那么它也同时能够接受数组和列表：

```
>>> points = np.array([[0,0,0],[1,0,0],[0,1,0],[0,0,1]], 'f')
>>> triangles = np.array([[0,1,3],[0,3,2],[1,2,3],[0,2,1]])
>>> values = np.array([1.1, 1.2, 2.1, 2.2])
>>> mesh = tvtk.PolyData(points=points, polys=triangles)
>>> mesh.point_data.scalars = values
>>> mesh.points
[(0.0, 0.0, 0.0), (1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]
>>> mesh.polys
<tvtk_classes.cell_array.CellArray object at 0x142D4F60>
>>> mesh.polys.to_array()
array([3, 0, 1, 3, 3, 0, 3, 2, 3, 1, 2, 3, 3, 0, 2, 1])
>>> mesh.point_data.scalars
[1.1000000000000001, 1.2, 2.1000000000000001, 2.2000000000000002]
```

第 10 章

Mayavi——更方便的可视化

虽然 VTK 可视化软件包的功能很强大，Python 的 TVTK 库也很方便简洁，但是用这些工具快速编写实用的三维可视化程序仍然是非常具有挑战性的。因此，基于 VTK 开发出了许多可视化软件，例如 ParaView、VTKDesigner、Mayavi 等。

Mayavi 完全用 Python 编写而成，它不但是一个方便实用的可视化软件，而且可以方便地用 Python 进行编写扩展，嵌入到用户编写的 Python 程序中，另外还提供了面向脚本的 mlab 模块，以方便用户快速绘制三维图形。

10.1 用 mlab 快速绘图

和 matplotlib 的 pylab 一样，Mayavi 的 mlab 模块提供了方便快捷的三维绘制函数。只要将数据准备好，通常只需要调用一次 mlab 模块的绘图函数，就可以看到数据的三维显示效果，非常适合在 IPython 中交互使用。

10.1.1 点和线

三维空间中独立的点使用 point3d()绘制，plot3d()是将一系列的点连接起来，从而绘制出三维曲线。它们可以有 3 或 4 个数组参数。这些数组的形状完全相同，前 3 个参数 x、y、z 对应点的 X 轴、Y 轴和 Z 轴的坐标值，如果有第 4 个参数 s，它指定每个坐标点对应的数值(标量值)。point3d()的第 4 个参数还可以是通过坐标计算标量值的函数。下面是这两个函数的调用格式：

```
plot3d(x, y, z, ...)
plot3d(x, y, z, s, ...)          #s 为保存每个点对应的标量值的数组

points3d(x, y, z...)
points3d(x, y, z, s, ...)        #s 为保存每个点对应的标量值的数组
points3d(x, y, z, f, ...)        #f 为计算每个点对应的标量值的函数
```

参数 x、y、z 决定了三维空间中每个点的坐标，而每个点对应的数值则可以用点的颜色、大小、线的粗细等直观地表现。

每个函数还有许多关键字参数，用来设置各种绘图属性，例如点的形状、线的宽度、颜色、颜色映射等等。所有这些参数能够设置的绘图属性，都可以在显示出图形窗口之后，在流水线

对话框中交互式地修改。因此本书不对这些参数进行详细讲解，读者可以阅读函数文档，了解每个关键字参数的含义。

我们以使用 plot3d()绘制洛伦茨吸引子轨迹为例，介绍如何绘制三维空间中的曲线，并且使用流水线对话框对图形的各种属性进行调整。下面是绘制洛伦茨吸引子轨迹的程序，算法请参照 3.4.2 节，默认的绘图效果如图 10-1 所示[①](见文前彩插)。

mlab_odeint_lorenz.py
用 plot3d 绘制洛伦茨吸引子轨迹

```
from scipy.integrate import odeint
import numpy as np

def lorenz(w, t, p, r, b):
    x, y, z = w
    return np.array([p*(y-x), x*(r-z)-y, x*y-b*z])

t = np.arange(0, 30, 0.01)
track1 = odeint(lorenz, (0.0, 1.00, 0.0), t, args=(10.0, 28.0, 3.0)) ❶

from enthought.mayavi import mlab ❷
mlab.plot3d(track1[:,0], track1[:,1], track1[:,2], t, tube_radius=0.2) ❸
mlab.show()
```

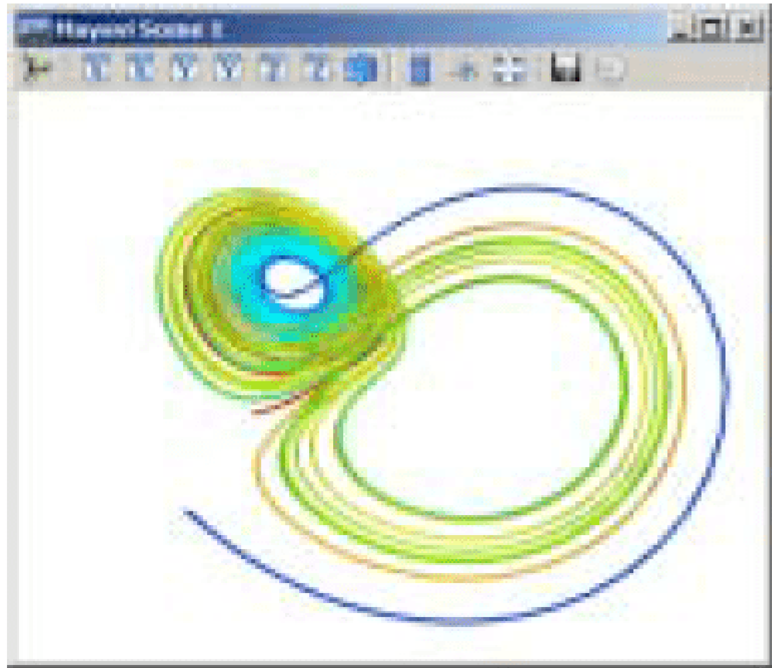


图 10-1　使用 plot3d 函数绘制的洛伦茨吸引子轨迹，曲线使用很细的圆管绘制

❶使用 odeint()得到的轨迹数据 track1 是一个二维数组，它的第 0 轴的长度是轨迹的点数，第 1 轴的长度为 3，分别为轨迹上每点的 X 轴、Y 轴和 Z 轴坐标。

❷载入 mlab 模块之后，调用❸plot3d()绘制轨迹曲线。其中，track1 为轨迹的坐标数组，将其拆分为三个轴上的分量之后，传递给 plot3d()进行绘图。用时间数组 t 作为标量值数组，因此轨迹上每个点对应的标量值就是到达此点的时间。tube_radius 参数用来设置曲线的粗细，曲线

① 为了更清楚地显示曲线，实际上用鼠标进行过放大和旋转操作。

实际上是采用极细的圆管绘制而成。



由于 mlab 模块的载入比较费时，建议在 IPython(mlab)或 IPython(wxPython)下用 run 命令执行绘图程序，这样既可以加速程序的重复运行，又能在 IPython 环境下观察各个对象的内容。

在三维场景中，可以使用键盘和鼠标对场景照相机或场景中的物体进行操作：

- 照相机模式：默认模式，在此模式下，所有的鼠标操作都是针对照相机进行的，按“C”键可切换到此模式。
- 角色模式：通过鼠标可以修改场景中物体的方向和位置，按“A”键可切换到此模式。

在照相机模式下：

- 旋转场景：使用鼠标左键拖动或者用键盘的方向键。
- 平移场景：使用鼠标中键进行拖动或者按住 Shift 键并使用左键拖动，抑或用 Shift+方向键。
- 缩放场景：使用鼠标右键上下拖动或者使用“+”和“-”键。
- 滚动照相机：按住 Ctrl 键并用左键拖动。

窗口中的工具栏还提供了从坐标轴的 6 个方向观察场景、等角投影、切换平行透视和成角透视的按钮。

10.1.2 Mayavi 的流水线

工具栏中最左边的图标是打开Mayavi pipeline 对话框[②]的按钮，单击此按钮将打开如图 10-2 所示的对话框。左边的 Pipeline 选项卡中用树状控件显示了构成场景的流水线。此流水线是 Mayavi 在 TVTK 的流水线之上进行封装的结果。选中流水线中的某个对象之后，窗口右侧部分将显示设置选中对象用的界面。让我们看看 plot3d()生成的流水线中都有哪些对象：

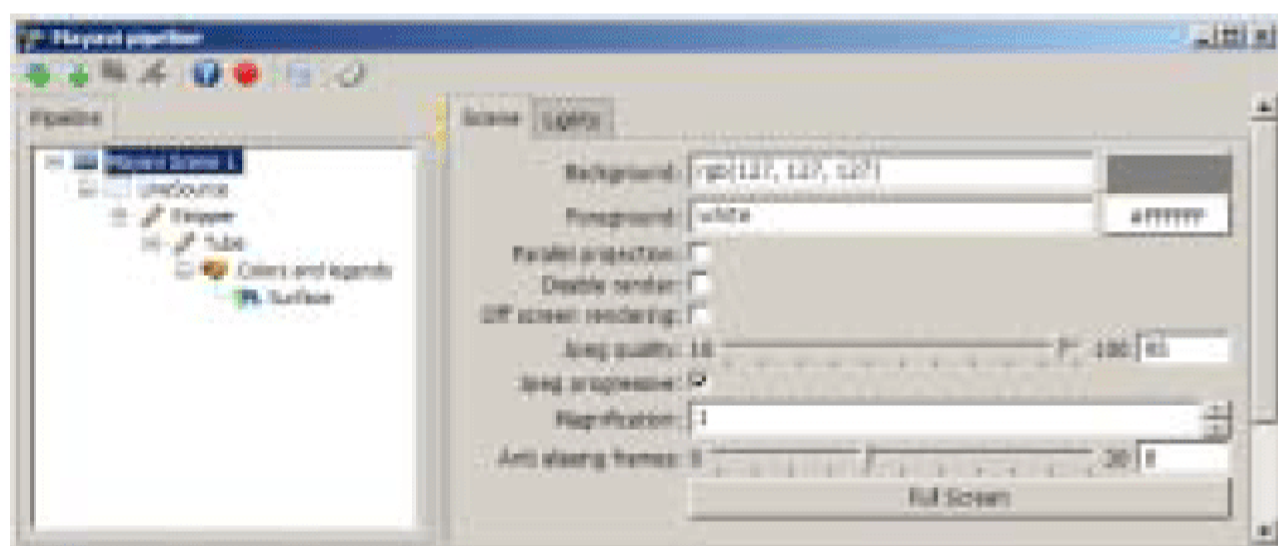


图 10-2　使用 plot3d 绘制的洛伦茨吸引子轨迹的流水线对话框

- Mayavi Scene 1：处于树的最顶层的对象表示场景。在其配置界面中可以设置场景的背景色和前景色、灯光以及其他一些选项。例如，将背景色“Background”改为白色，

② 后面统一称之为流水线对话框。

将前景色“Foreground”改为灰色。可以在 IPython 中输入下面的语句，获取场景对象的背景色：

```
>>> s = mlab.gcf() # 首先获得当前的场景
>>> s
<enthought.mayavi.core.scene.Scene object at 0x089D4450>
>>> s.scene.background
>>> (1.0, 1.0, 1.0)
```



为了节省篇幅，书中没有列出本例涉及的各个配置界面的截图。请读者对照程序的运行界面阅读以下说明。

- LineSource：线数据源。在其配置界面中，第一个项目为每个点对应的标量数据的名称，在本例中只有一个名为 scalars 的标量数据，它就是我们传递给 plot3d()的第 4 个数组——表示轨迹中每点时间的数组 t。下面的语句从场景中获取 LineSource 对象，并且获取其中的各种数据：

```
>>> source = s.children[0] # 获得场景的第一个子节点，也就是 LineSource
>>> source
<enthought.mayavi.sources.vtk_data_source.VTKDataSource object at 0x08726750>
>>> source.name # 节点的名字，也就是流水线中显示的文字
'LineSource'
>>> source.data.points # LineSource 中的坐标点
>>> [(0.0, 1.0, 0.0), ...,
(-1.1439147650081221, -1.9822077918751035, 11.001854100552031)],
length = 3000
>>> #每个点对应的标量数组，它和时间数组 t 共享数据内存
>>> source.data.point_data.scalars
>>> [0.0, ..., 29.99], length = 3000
```

- Stripper：根据其内部 filter 对象的 maximum_length 属性，对线数据源进行分段处理。在本例中，输入的线数据源有 3000 个点，而 maximum_length 属性为 1000，即每 1000 个点将对应一条线。因此 stripper 对象输出的 PolyData 对象中有 3 条线：

```
>>> stripper = source.children[0]
>>> stripper.filter.maximum_length
1000
>>> stripper.outputs[0].number_of_points
3000
>>> stripper.outputs[0]
<tvtk_classes.poly_data.PolyData object at 0x0A5E2F30>
>>> stripper.outputs[0].number_of_lines
3
```

- Tube：将输入的 PolyData 数据中的每条线转换为表示三维圆管的 PolyData 数据。它的配置界面中有许多参数可以改变其生成圆管的方式。例如，将“Vary radius”设置为“vary_radius_by_scalar”，则圆管的粗细由每个点对应的标量值决定。而加粗的比例则由“Radius factor”参数决定，我们将其设置为 3，于是此圆管最粗处(终点)的半径是最细处(起点)的 3 倍。下面的语句获得 Tube 对象，并查看其输出对象的类型：

```
>>> tube = stripper.children[0] # 获得 Tube 对象
>>> tube.outputs[0] # tube 的输出是一个 PolyData 对象，它是一个三维圆管
<tvtk_classes.poly_data.PolyData object at 0x089E9B70>
```

- Colors and legends：在其配置界面的“Scalar LUT”选项卡中可以设置标量值转换为颜色的查询表(Look Up Table)。例如，将“Lut mode”改为 Blues，于是场景中圆管的颜色变成了从白色到深蓝色的渐变。选中“Show legend”选择框，场景中便添加了一个颜色条，用来显示颜色和标量值之间的关系。下面的语句获取我们所选择的颜色查询表：

```
>>> manager = tube.children[0]
>>> manager.scalar_lut_manager.lut_mode
'Blues'
```

- Surface：将 Tube 输出的 PolyData 数据转换为最终在场景中显示的三维实体。通过其配置界面的“Actor”选项卡，可以对实体进行配置。例如，将“Representation”选择为“wireframe”，并将 Line width 设置为 0，实体将采用细线框模型显示。将“Opacity”设置为 0.6，实体则变为半透明状态。下面的语句获取我们刚刚修改的这两个配置：

```
>>> surface = manager.children[0]
>>> surface.actor.property.representation
>>> 'wireframe'
>>> surface.actor.property.opacity
>>> 0.59999999999999998
```

修改之后的场景如图 10-3 所示。

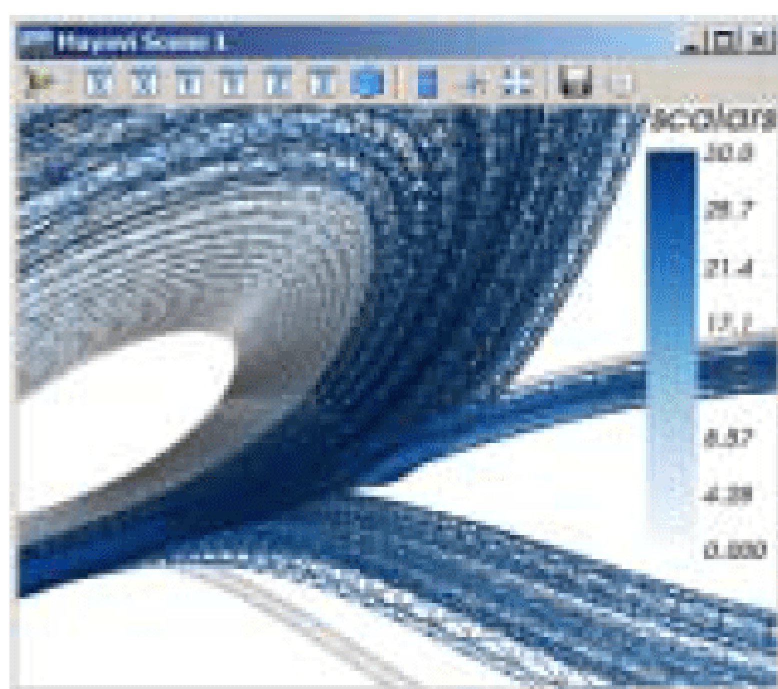


图 10-3　在流水线对话框中修改许多配置之后的洛伦茨吸引子轨迹

通过前面获取流水线中各个对象配置的程序，我想读者应该掌握了用程序配置这些属性的方法：

- 先获得场景对象，例如使用 mlab.gcf()。
- 通过每个对象的 children 属性，在流水线中找到需要修改的对象。
- 当其配置窗口有多个选项卡或多个配置分组框时，意味着其属性可能需要一级一级地获得。对象的属性名和界面上的文字之间存在很简单的转换关系：首字母变大写、下划线变空格。例如 Surface 对象的 Actor 选项卡，Property 分组框中的“Line width”选项用程序描述就是：

```
>>> surface.actor.property.line_width
```

Mayavi 还提供了脚本录制功能，以方便我们编写配置各种属性的程序。单击流水线对话框工具栏中的红色圆形图标即可开始脚本录制，同时将打开一个脚本对话框。后面的界面配置操作都会被记录到此脚本对话框中。

10.1.3 二维图像的可视化

三维空间中的曲面可以用 surf()进行绘制，这实际上是将二维图像(数组)绘制成三维曲面，用曲面的高度表示图像中每点的值。下面是绘制 2.2.4 节中图 2-5 的程序。



mlab_surf.py
用 surf()绘制三维空间中的曲面

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j] ❶
z = x * np.exp( - x**2 - y**2) ❷

face = mlab.surf(x, y, z, warp_scale=2) ❸
mlab.axes(xlabel='x', ylabel='y', zlabel='z') ❹
mlab.outline(face)
mlab.show()
```

图 10-4 显示了 surf()绘制的曲面及其流水线对话框中的内容(见文前彩插)。程序中，❶先通过 ogrid 对象计算两个形状分别为(20,1)和(1,20)的数组 x 和 y。❷然后通过广播运算，计算出由 x 和 y 构成的等距网格上每点的函数值 z，z 是一个形状为(20,20)的数组。❸接着调用 mlab.surf()将数组 z 绘制成三维空间中的曲面，所绘制的曲面在 X-Y 平面上的投影是一个等距离的网格。❹最后通过 mlab.axes()和 mlab.outline()，分别在三维场景中添加坐标轴和曲面区域的外框。

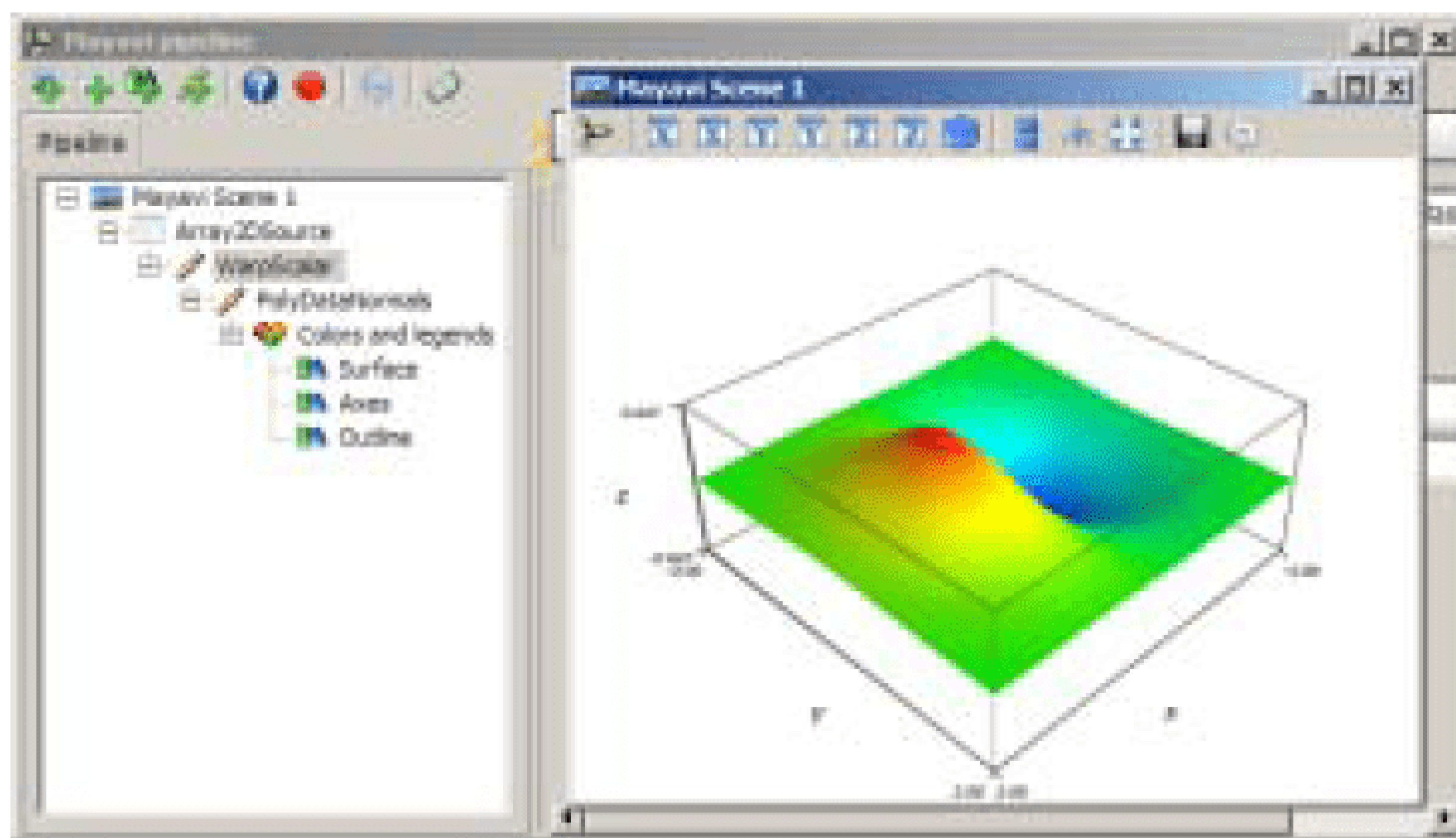


图 10-4　surf()绘制的曲面及其流水线对话框



和 matplotlib 相反，在 Mayavi 中，二维数组的第 0 轴表示 X 轴，第 1 轴表示 Y 轴。

仔细观察曲面的颜色就会发现，颜色和曲面的高度有一一对应的关系，曲面上的点越高，其颜色越红，越低则颜色越蓝。

打开流水线对话框，可以看到数据源是一个 Array2DSource 对象，它的输出是一个 TVTK 的 ImageData 对象：

```
>>> data = mlab.gcf().children[0]
>>> img = data.outputs[0]
>>> img
<tvtk_classes.image_data.ImageData object at 0x1097F1B0>
```

ImageData 对象是一个表示三维图像的数据集。在 ImageData 对象中，实际上不保存空间中每点的坐标，而是通过 origin、spacing 和 dimensions 等属性计算出一个三维空间中的等距离网格，网格中每个点对应的标量值保存在 point_data.scalars 中，这些值决定了曲面的高度和颜色。

```
>>> img.origin # X 轴、Y 轴、Z 轴的起点
array([-2., -2.,  0.])
>>> img.spacing # X 轴、Y 轴、Z 轴上点的间隔
array([ 0.21052632,  0.21052632,  1.        ])
>>> img.dimensions # X 轴、Y 轴、Z 轴上点的个数
array([20, 20,  1])
>>> img.point_data.scalars # 每个点对应的标量值
[-0.000670925255805, ..., 0.000670925255805], length = 400
```

通过上面的分析可以看出，surf()的功能是将一个二维图像转换为三维空间中的曲面。因此，曲面上每个点的 X 轴、Y 轴的坐标都是通过网格配置计算出来的。

由于曲面的高度与其在 X-Y 平面上的尺寸可能相差很大，因此在流水线中，在 Array2DSource 对象的下面是一个 WarpScalar 对象，它将输入数据沿着 Z 轴方向进行缩放，可以看到其配置面板中的“Scale factor”为 2，它是由 surf()的 warp_scale 参数决定的。WarpScalar 对象的输出是一个 PolyData 对象：

```
>>> data.children[0].outputs[0]
<tvtk_classes.poly_data.PolyData object at 0x11329960>
```

流水线中剩下的对象请读者自己进行研究，通过研究流水线可以了解到 Mayavi 内部的组织构造，这有助于我们创建自己的流水线结构，进而对复杂的数据进行可视化。

如果数据在三个坐标轴上的范围相差很大，在进行可视化时就需要调整坐标轴的显示比例，以达到更好的可视化效果。例如在下面的曲面函数中，X 轴方向需要更大的显示范围：

```
>>> x, y = np.ogrid[-10:10:100j, -1:1:100j]
>>> z = np.sin(5*((x/10)**2+y**2))
```

如果直接使用数据的范围进行显示，效果如图 10-5(左)所示(见文前彩插)。虽然可以很直观地看出 X 轴的显示范围是 Y 轴的 10 倍，但是却很难观察曲面的一些细节信息。

```
>>> mlab.surf(x, y, z)
>>> mlab.axes()
```

通过 surf()的 extent 参数可以修改坐标轴的数据范围：

```
>>> mlab.surf(x, y, z, extent=(-1,1,-1,1,-0.5,0.5))
>>> mlab.axes(nb_labels=5)
```

extent 参数是一个包含 6 个元素的序列，分别用来指定：X 轴最小值、X 轴最大值、Y 轴最小值、Y 轴最大值、Z 轴最小值、Z 轴最大值。这些值将修改数据范围，因此所绘制曲面的 X 轴、Y 轴的范围相同(见文前彩插)，而曲面在 Z 轴上的高度是 X 轴、Y 轴范围的一半，效果如图 10-5(中)所示(见文前彩插)。由于 extent 参数改变的是数据的范围，因此坐标轴上的刻度值也随之发生变化。为了解决这个问题，可以给 axes()传递 ranges 参数：

```
>>> mlab.axes(ranges=(x.min(),x.max(),y.min(),y.max(),z.min(),z.max()), nb_labels=5)
```

ranges 参数也是一个包含 6 个元素的序列，分别指定三个坐标轴上的刻度范围，效果如图 10-5(右)所示(见文前彩插)。

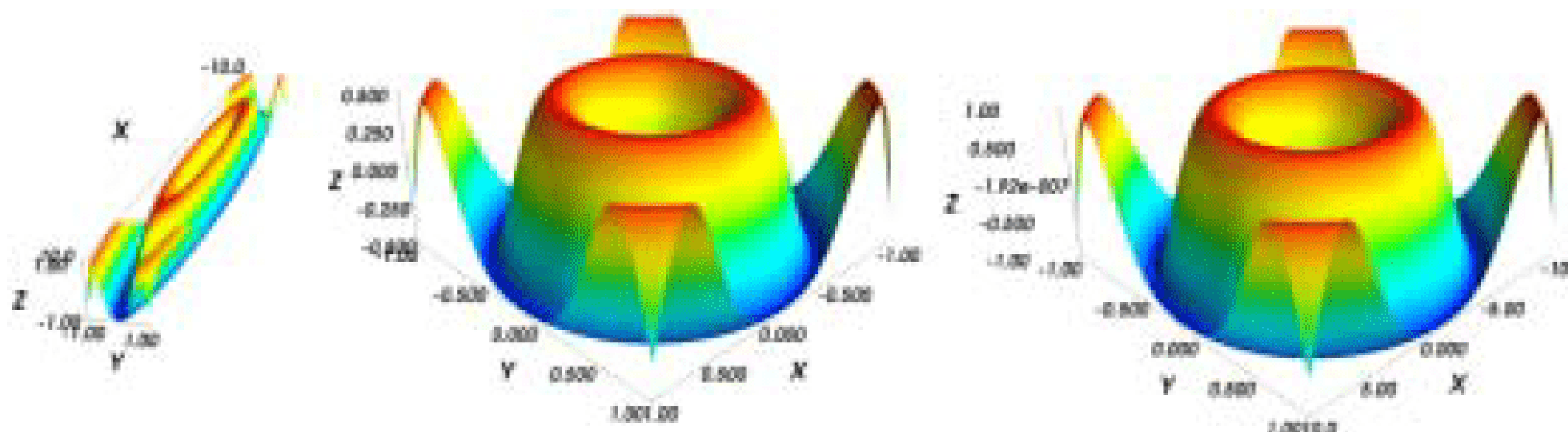


图 10-5　修改坐标轴的显示比例

除了 surf()之外，imshow()和 contour_surf()也是可视化二维图像的有效工具。imshow()将二维图像放在三维空间中显示，它只有一个二维数组参数，效果和将 surf()的 warp_scale 参数设置为 0 的效果一样。下面的语句将二维数组 z 用 imshow()绘制成图，效果如图 10-6(左)所示(见文前彩插)。



mlab_imshow_contour.py
显示二维图像和等高线

```
>>> mlab.figure()
>>> mlab.imshow(z)
```

contour_surf()和 surf()的参数类似，但可以通过 contours 参数指定等高线的数目或等高值的列表。下面的语句将曲面以 20 条等高线表示，如图 10-6(右)所示(见文前彩插)。

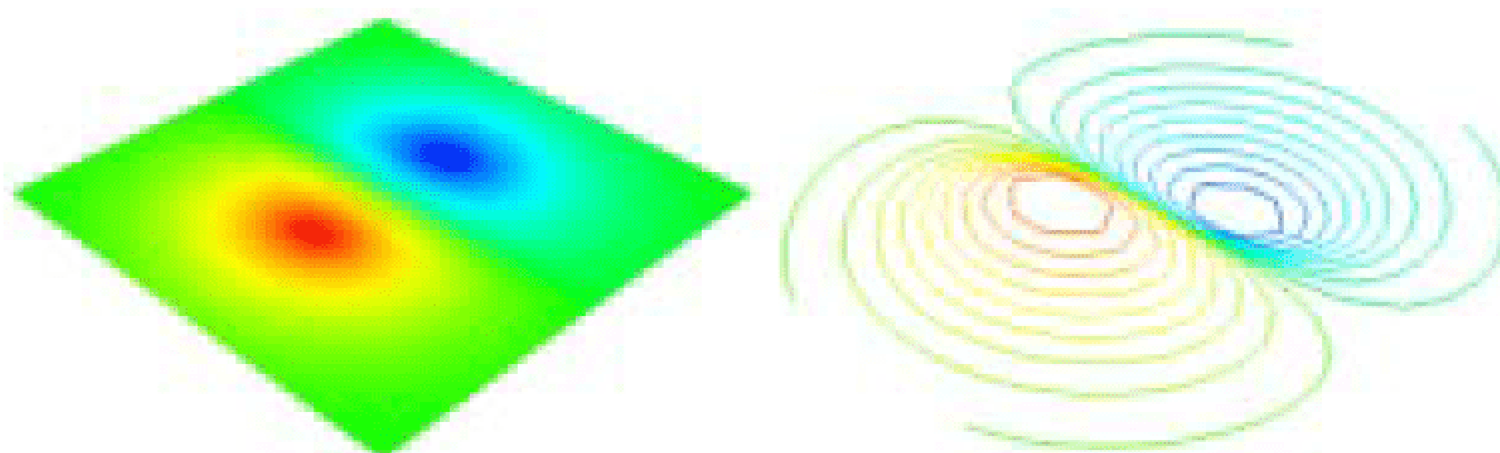


图 10-6　用 imshow 绘制图像(左)、用 contour_surf 绘制等高线(右)

```
>>> mlab.figure()
>>> mlab.contour_surf(x,y,z,warp_scale=2,contours=20)
```

实际上，在用 surf()绘制曲面之后，在流水线对话框中对 Surface 对象进行如下配置，也可以实现和 contour_surf()一样的效果：

- 在“Contours”选项卡中，选中“Enable Contours”。
- 选中“Auto contours”选项，并且指定“Number of contours”为 20，这样会自动产生 20 条等高线。

- 也可以不选中“Auto contours”选项，然后手工添加等高线。

或者用下面的代码进行等高线设置，其中，变量 face 为 surf()返回的对象，也就是流水线中的 Surface：

```
>>> face.enable_contours = True
>>> face.contour.number_of_contours = 20
```

10.1.4　网格面

如果需要绘制更复杂的三维曲面，可以使用 mesh()。下面是使用 mesh()绘制复杂曲面的程序，效果如图 10-7 所示(见文前彩插)。

Mlab_mesh.py
用 mesh 函数快速绘制复杂的 3D 曲面

```
from numpy import pi, sin, cos, mgrid
from enthought.mayavi import mlab

# 创建数据
dphi, dtheta = pi/80.0, pi/80.0
[phi,theta] = mgrid[0:pi+dphi*1.5:dphi,0:2*pi+dtheta*1.5:dtheta]
m0, m1, m2, m3, m4, m5, m6, m7 = 4,3,2,3,6,2,6,4
r = (sin(m0*phi)**m1 + cos(m2*phi)**m3 +  ❶
    sin(m4*theta)**m5 + cos(m6*theta)**m7)

x = r*sin(phi)*cos(theta) ❷
y = r*cos(phi)
z = r*sin(phi)*sin(theta)

# 使用 mesh()绘制网格面
s = mlab.mesh(x, y, z) ❸

mlab.show()
```

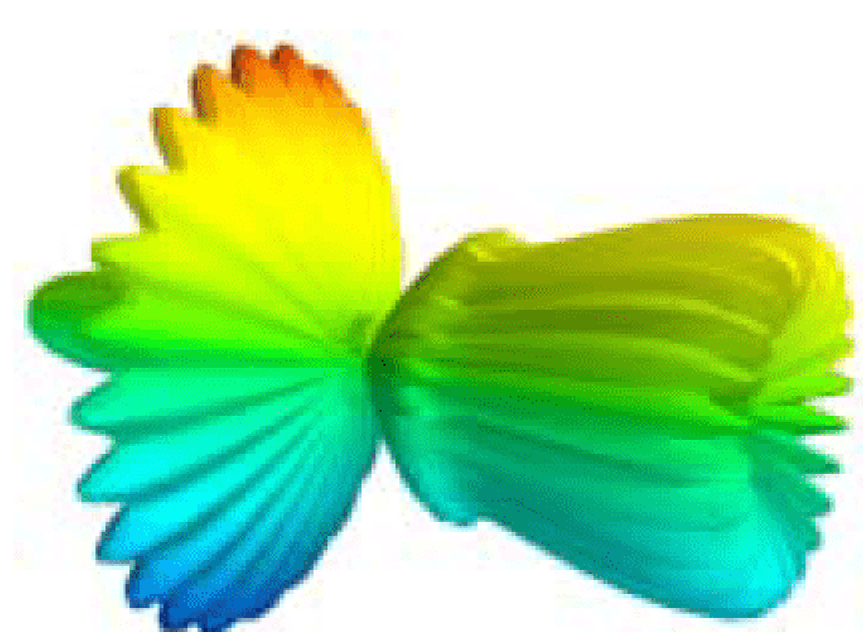


图 10-7　使用 mesh 函数绘制的 3D 旋转体

❸程序中调用 mesh()绘制曲面，它和 surf()类似，三个数组参数 x、y、z 都是形状相同的二维数组。这些数组中相同下标的三个数值组成曲面上某点的三维坐标。点之间的连接关系(边和面)由它们在 x、y、z 数组中的位置关系决定。❶曲面上各点的坐标在球坐标系中计算，❷然后按照坐标转换公式将球坐标系转换为笛卡尔坐标系。

为了方便读者理解 mesh()是如何绘制出曲面的，下面通过手工输入坐标的方式，绘制如图 10-8 所示的立方体表面的一部分(见文前彩插)。



mlab_mesh_cube.py
用 mesh()绘制手工指定顶点坐标的立方体

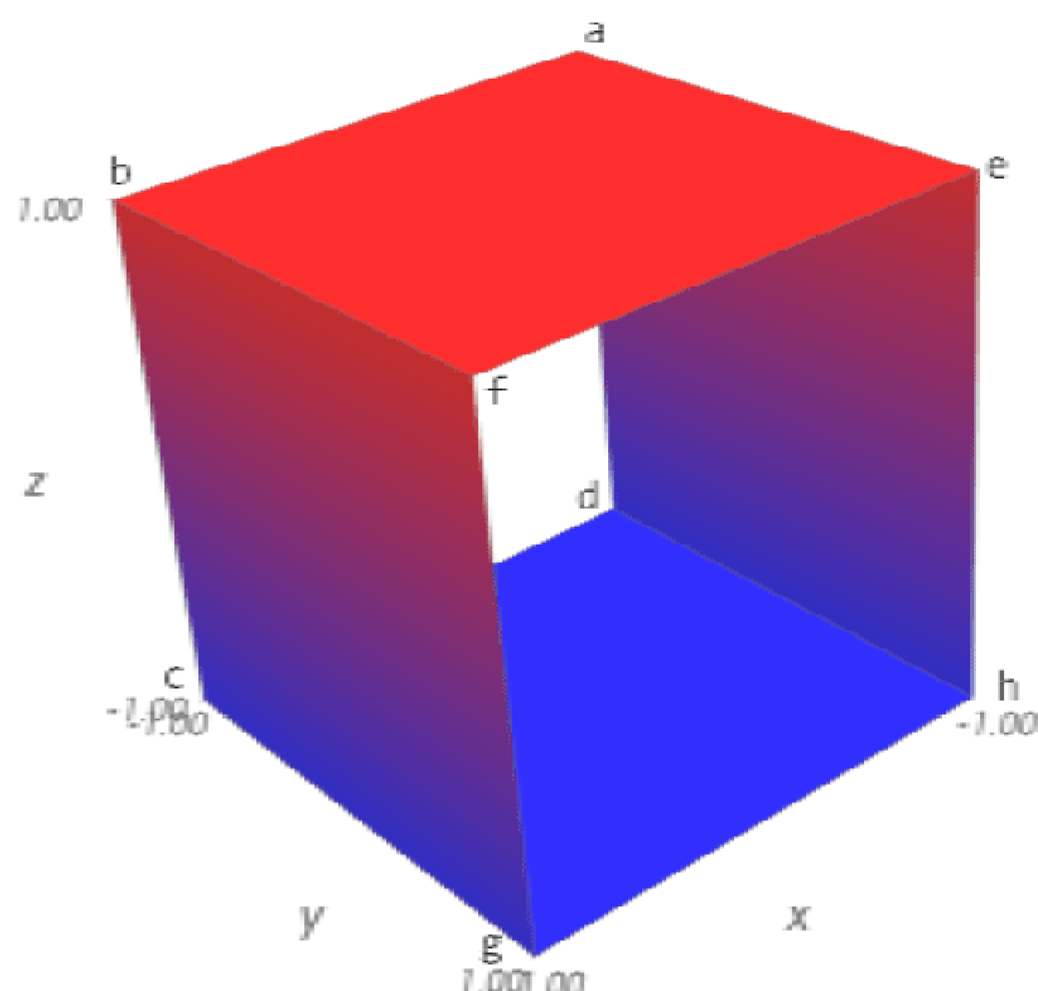


图 10-8　组成立方体的各个面和顶点坐标

传递给 mesh()的三个数组 x、y、z 的内容如下：

```
x = [[-1,1,1,-1,-1],
     [-1,1,1,-1,-1]]

y = [[-1,-1,-1,-1,-1],
     [ 1, 1, 1, 1, 1]]

z = [[1,1,-1,-1,1],
     [1,1,-1,-1,1]]
```

数组中下标相同的三个数字组成一个三维坐标，因此这三个数组实际描述的坐标点为：

```
[
   [(-1, -1, 1), (1, -1, 1), (1, -1, -1), (-1, -1, -1), (-1, -1, 1)],
   [(-1,  1, 1), (1,  1, 1), (1,  1, -1), (-1,  1, -1), (-1,  1, 1)]
]
```

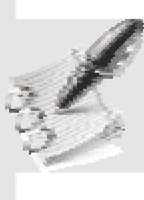
为了理解方便，图中将上面的坐标点用字母来表示：

```
[[a,b,c,d,a],
 [e,f,g,h,e]]
```

坐标点之间的关系由它们在数组中的位置决定，因此下面 4 组坐标点将构成 4 个正方形平面：

```
a, b, f, e -> 顶面
b, c, g, f -> 左面
c, d, h, g -> 底面
d, a, e, h -> 右面
```


使用 mesh()可以很方便地将二维平面上的曲线绕着对称轴进行旋转，从而得到旋转面。下面的程序用 mesh()绘制由抛物线旋转之后得到的旋转抛物面，如图 10-9 所示(见文前彩插)。


mlab_surface_of_revolution.py
用 mesh()绘制旋转抛物面


```
import numpy as np
from enthought.mayavi import mlab

rho, theta = np.mgrid[0:1:40j, 0:2*np.pi:40j] ❶

z = rho*rho ❷

x = rho*np.cos(theta) ❸
y = rho*np.sin(theta) ❸

s = mlab.mesh(x,y,z)
mlab.show()
```

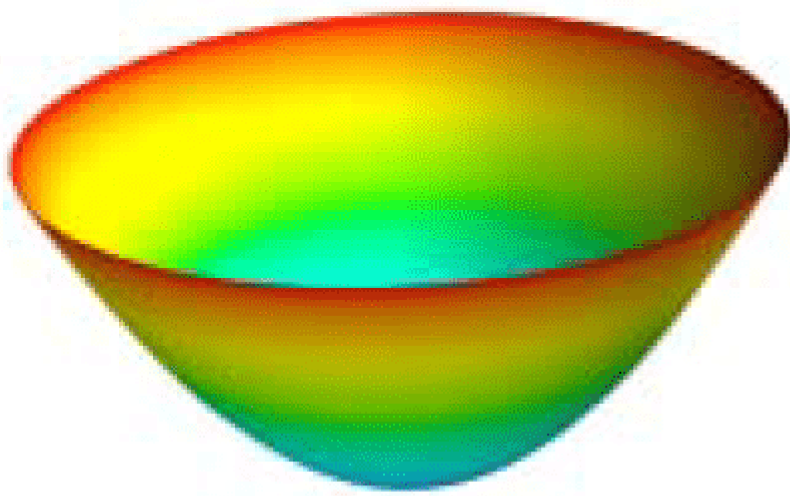


图 10-9　用 mesh()绘制旋转抛物面

旋转抛物面上点的坐标很容易在圆柱坐标系(ρ,ϕ,z)中计算得出。❶首先在(ρ,ϕ)平面中创建一个 40 × 40 的二维网格。❷通过ρ计算出抛物面上每点的高度z。由于是旋转面，因此高度

和ϕ无关。❸将圆柱坐标系转换为直角坐标系，得到 mesh()所需的三个数组。

使用 mesh()还可以绘制出前面 surf()所绘制的曲面。下面是用 mesh()绘制曲面的程序：



mlab_mesh_surf.py
用 mesh()绘制函数曲面

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.mgrid[-2:2:20j, -2:2:20j] ❶
z = x * np.exp( - x**2 - y**2)
z *= 2
c = 2*x + y ❷

pl = mlab.mesh(x, y, z, scalars=c) ❸
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(pl)
mlab.show()
```

❶这里使用 mgrid 对象产生数组 x 和 y。这是因为用 mesh()绘制曲面时，必须给出曲面上每点的坐标值。因此参数 x 和 y 不能是 ogrid 产生的形状为(1,N)和(M,1)的数组，而必须都是形状为(M,N)的数组。

❷为了演示 mesh()绘制曲面的优点，我们另外计算一个二维数组 c，❸并且把它传递给 mesh()的 scalars 参数。于是曲面上每个点对应的标量值都将使用数组 c 中相应下标的值。这样可以使用数组 c 对曲面上的每个点进行着色，得到一个颜色和高度无关的曲面，相对 surf()绘制的曲面，它能表达更多的信息。图 10-10 为程序的绘制效果及其对应的流水线(见文前彩插)。

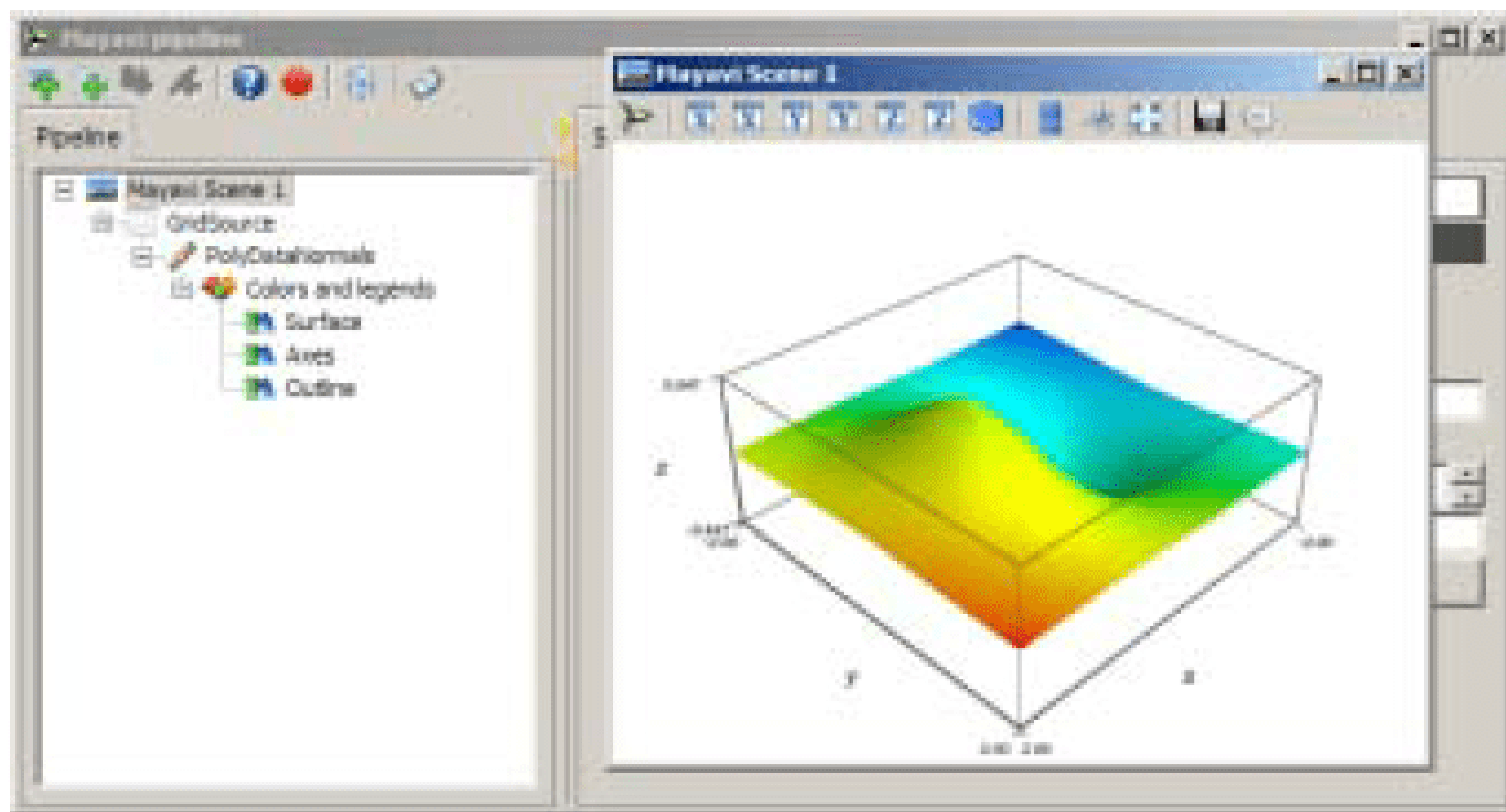
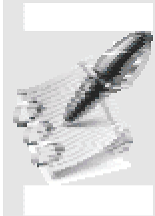


图 10-10　用 mesh()绘制高度和颜色不同的曲面

10.1.5 修改和控制流水线

用 surf()也可以绘制高度和颜色不同的曲面，但是需要对流水线进行一些改动。



mlab_surf_color.py
通过修改 surf()的流水线绘制高度和颜色不同的曲面

在 IPython(mlab)中运行前面介绍过的绘制曲面的程序：

```
>>> run mlab_surf.py
```

我们需要为每个点添加一个新的标量值，以控制它们的颜色。因此，首先获得流水线中“Array2DSource”对象内部的 ImageData 对象[③]：

```
>>> img = mlab.gcf().children[0].image_data
```

然后给 img 添加一个标量数组，并且命名为"color"。注意数组 c 的第 0 轴对应于 X 轴，而第 1 轴对应于 Y 轴，因此需要将其转置，ravel()方法则将二维数组转换为一维数组：

```
>>> c = 2*x + y # 表示颜色的标量数组
>>> array_id = img.point_data.add_array(c.T.ravel())
>>> img.point_data.get_array(array_id).name = "color"
>>> img.point_data.update()
```

如果读者对为什么需要转置有疑问，查看一下数组 z 在 ImageData 对象中的保存顺序，便可以看出，二维数组 z 的数据是按照第 0、1 轴的顺序保存在 scalars 数组中的：

```
>>> z[:3,:3] # 原始的二维数组中的元素
array([[-0.00067093, -0.00148987, -0.00302777],
       [-0.00133304, -0.00296016, -0.00601578],
       [-0.00239035, -0.00530804, -0.01078724]])
>>> # ImageData 中标量值的顺序
>>> img.point_data.scalars.to_array()[:3]
>>> array([-0.00067093, -0.00133304, -0.00239035]) # 和数组 z 中第 0 列的数值相同
```

接下来需要在流水线的“PolyDataNormals”和“Colors and legends”之间插入一个 SetActiveAttribute 对象，它将 PolyDataNormals 的输出数据中名为"color"的标量值设置为当前标量值。下面先获得 PolyDataNormals 对象：

```
>>> normals = mlab.gcf().children[0].children[0].children[0]
```

③ 流水线中的“Array2Dsource”实际上是一个 ArraySource 对象，查看它的源代码可知，它使用 image_data 属性保存所创建的 ImageData 对象。

通过下面的语句可以看到，“PolyDataNormals”输出的 PolyData 对象的当前标量值为数组 z 中的数据：

```
>>> normals.outputs[0].point_data.scalars.to_array()[:3]
array([-0.00067093, -0.00133304, -0.00239035])
```

接下来是插入操作。首先获得 normals 的下一级对象，并将其从 children 列表中删除：

```
>>> surf = normals.children[0]
>>> del normals.children[0]
```

然后调用 pipeline.set_active_attribute()，创建一个 SetActiveAttribute 对象，并将其添加到 normals.children 列表中。通过 point_scalars 参数将名为"color"的数组设置为默认标量值：

```
>>> active_attr = mlab.pipeline.set_active_attribute(normals, point_scalars="color")
```

最后将 surf 对象添加到 SetActiveAttribute 对象的子列表中：

```
>>> active_attr.children.append(surf)
```

现在，“PolyDataNormals”输出的 PolyData 对象的当前标量值已经变为数组 c 中的数据了，即它的当前标量值通过 SetActiveAttribute 对象被修改为名为"color"的数组：

```
>>> normals.children[0].outputs[0].point_data.scalars.to_array()[:3]
>>> array([-6.        , -5.57894737, -5.15789474])
```

于是后面的颜色查询表将使用数组 c 的值作为输入，从而使得曲面的高度和颜色分别使用不同的数据进行描绘。最终的绘图效果和相应的流水线如图 10-11 所示(见文前彩插)。

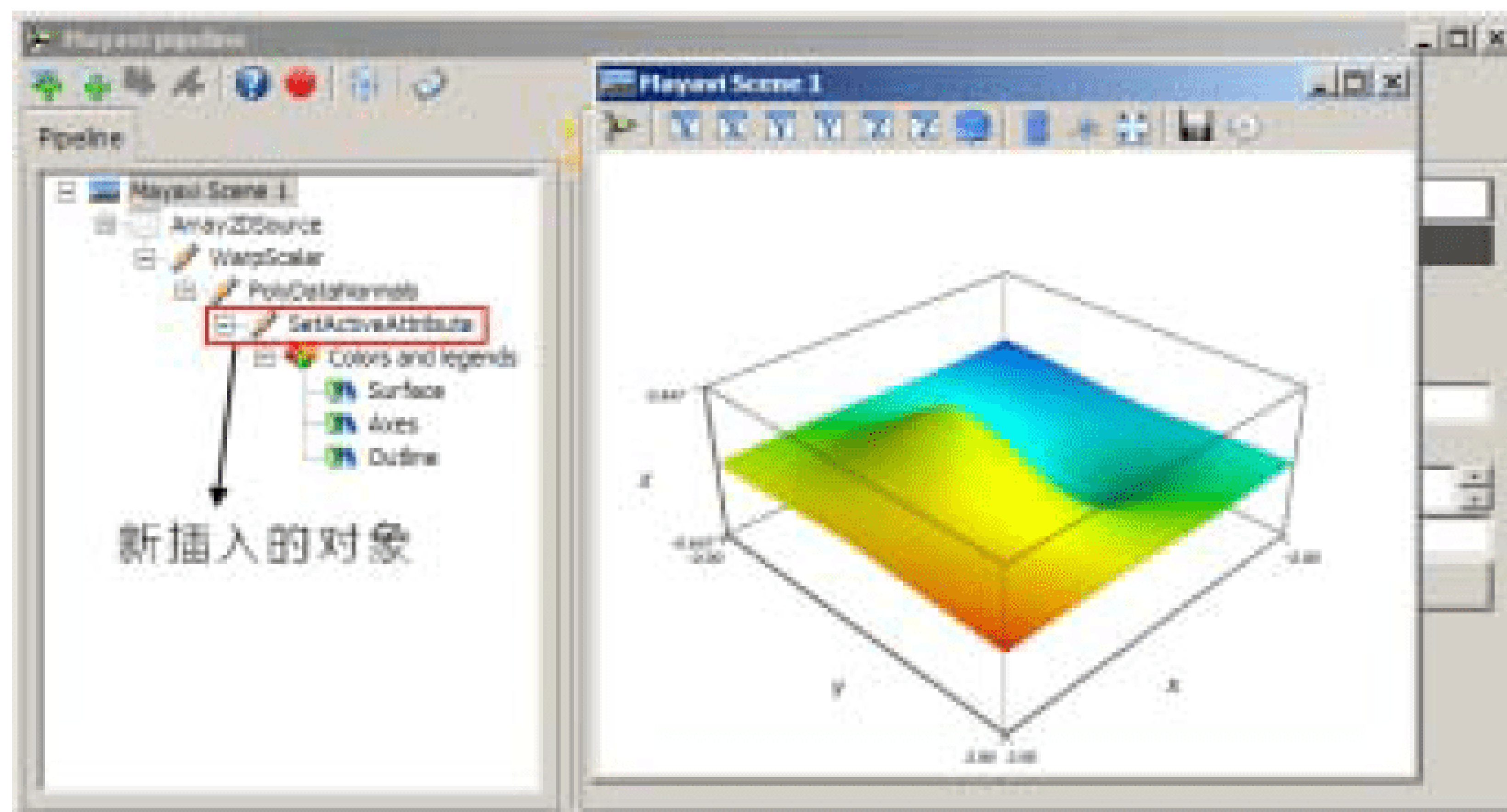


图 10-11　用 surf()绘制高度和颜色不同的曲面

也可以不调用 surf()，而直接创建流水线中的每个对象。完整的程序如下：



mlab_surf_color2.py
通过直接创建流水线来绘制高度和颜色不同的曲面

```
import numpy as np
from enthought.mayavi import mlab

# 创建数据
x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2) # 高度数据
c = 2*x + y # 颜色数据

src = mlab.pipeline.array2d_source(x, y, z)
dataset = src.mlab_source.dataset # 和 src.outputs[0]相同
array_id = dataset.point_data.add_array(c.T.ravel())
dataset.point_data.get_array(array_id).name = "color"
dataset.point_data.update()

# 创建流水线
warp = mlab.pipeline.warp_scalar(src, warp_scale=2.0)
normals = mlab.pipeline.poly_data_normals(warp)
active_attr = mlab.pipeline.set_active_attribute(normals,
    point_scalars="color")
surf = mlab.pipeline.surface(active_attr)

mlab.axes()
mlab.outline()
mlab.show()
```

直接创建流水线需要开发者对 Mayavi 流水线上的各种对象十分了解，因此建议读者首先熟悉 Mayavi 的界面操作以及各种对象的作用，然后通过录制脚本逐步学习。

10.1.6 标量场

前面介绍了如何对一维和二维数据进行可视化，下面看看三维数据的可视化问题。最简单的三维数据就是三维图像，它可以用一个三维数组表示。图像中每个点的三维坐标都由它在数组中的下标决定，而每个点对应的标量值则是数组中对应元素的值。这种三维图像可以用来描述标量场，例如房间中的温度分布、材料的密度分布，以及采用 X 射线断层成像(CT)技术采集到的人体的内部构造。

标量场有三种可视化方法：

- 等值面：和二维图像的等高线类似，用标量值相等的曲面，显示标量场的形态。
- 体素呈像：利用透明度和颜色直接呈现标量场的形态。

● 切面：通过对标量场进行切面处理，显示在某个切面上标量场的形态。



使用静态截图很难如实地表现可视化效果，为了更深入地理解这些可视化工具，请读者在计算机上运行本章的程序，并调整场景照相机，从各个方向进行观察。



mlab_scale_field.py
使用等值面、体素呈像和切面可视化标量场

我们用下面的程序计算一个含有两个点电荷的电势场，两个点电荷分别位于(-1, 0, 0)和(1, 0, 0)处，为了方便显示，我们只计算 Z 轴上(-2,0)区间的电势场的值：

```
>>> x, y, z = np.ogrid[-2:2:40j, -2:2:40j, -2:0:40j]
>>> s = 2/np.sqrt((x-1)**2 + y**2 + z**2) + 1/np.sqrt((x+1)**2 + y**2 + z**2)
```

使用 contour3d()可以快速绘制此电势场的等值面：

```
>>> surface = mlab.contour3d(s)
```

默认情况下，contour3d()绘制 5 个等分标量值范围的等值面，它不能很好地显示整个电势场的结构，因此我们用下面的语句修改等值面的范围、数目及透明度等属性：

```
>>> surface.contour.maximum_contour = 15 # 等值面的上限值为 15
>>> surface.contour.number_of_contours = 10 # 在最小值到 15 之间绘制 10 个等值面
>>> surface.actor.property.opacity = 0.4 # 透明度为 0.4
```

这些属性也可以在流水线对话框中交互式地进行修改，程序的绘制效果如图 10-12 所示(见文前彩插)。

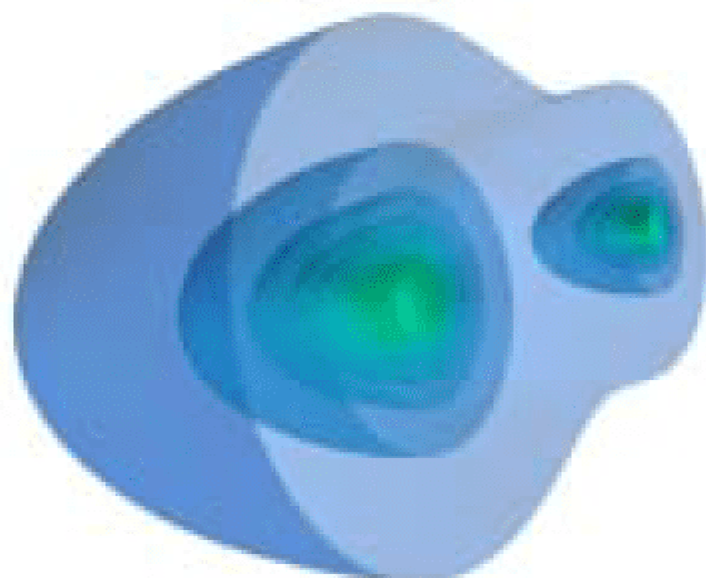


图 10-12　用等值面可视化电势场

使用等值面对标量场进行可视化时，外面的等值面可能会完全包含内部的等值面，观察不到内部的状态。例如在本例中，如果将 Z 轴的计算范围改为(-2, 2)，并且不设置等值面的透明度，那么无论绘制多少个等值面，都只能看到最外层的等值面。

体素呈像法用每个点的颜色和透明度对整个标量场进行润色，从而能够呈现更多的信息。体素呈像没有对应的函数，需要我们自己创建流水线：

```
>>> field = mlab.pipeline.scalar_field(s)
>>> mlab.pipeline.volume(field)
```

首先通过 scalar_field()在流水线中创建一个标量场数据源，然后通过 volume()将此数据源

用体素呈像进行可视化，效果如图 10-13(左)所示(见文前彩插)。

由于电势强度随着距离的平方衰减，因此整体的润色效果并没有突出电势强的部分。为了解决这个问题，可以给 volume()传递两个关键字参数——vmin 和 vmax。它们指定标量值的润色范围，即只绘制标量值在 vmin 到 vmax 之间的区域：

```
>>> mlab.pipeline.volume(field, vmin=1.5, vmax=10)
```

效果如图 10-13(右)所示，它很清楚地呈现出了电荷附近的电势情况(见文前彩插)。

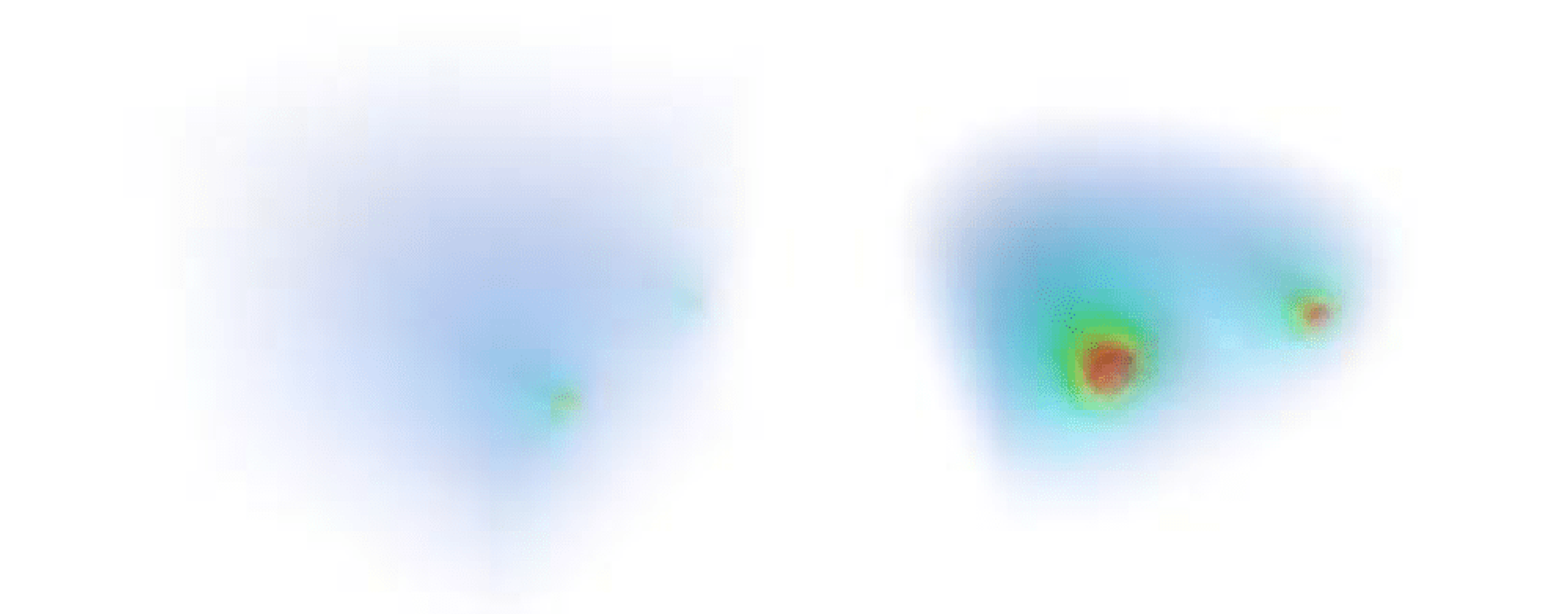


图 10-13　用体素呈像法可视化电势场：(左)默认效果，(右)通过 vmin 和 vmax 指定电势值的润色范围

我们还可以使用切片工具观察标量场在某个平面之上的数据，通常将切面工具和其他的工具同时使用，并且可以直接在三维场景中交互式地改变平面的位置和方向。下面的代码在流水线中添加一个标量切面，在流水线中它是“Colors and legends”的子节点。通过 plane_orientation 参数指定切面的法线方向为 Y 轴，即切面和 Y 轴垂直：

```
>>> cut = mlab.pipeline.scalar_cut_plane(field.children[0], plane_orientation="y_axes")
```

然后通过下面的代码设置切面工具的一些属性：

```
>>> cut.enable_contours = True # 开启等高线显示
>>> cut.contour.number_of_contours = 40 # 等高线的数目为 40
```

切面工具的效果如图 10-14 所示(见文前彩插)，图中还显示了添加切面工具之后的流水线。在 3D 场景中可以对切面工具进行如下操作：

- 拖动切面的红色外框可修改切面的位置。
- 拖动切面的法线箭头可修改切面的方向。
- 拖动法线和切面相交的灰色圆球，可改变切面的旋转中心。

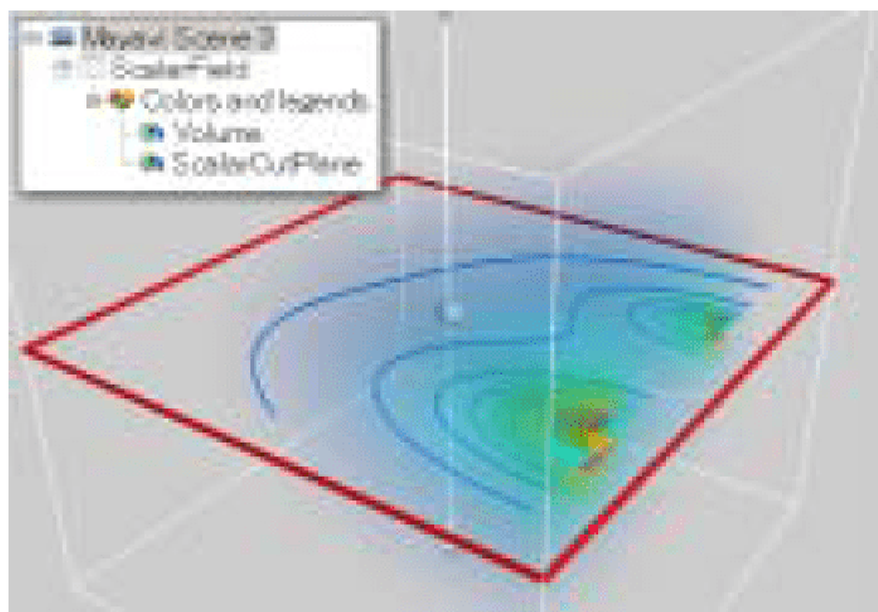


图 10-14　用切面工具观察电势场

10.1.7　矢量场

如果场中每一点的属性都可以用矢量来表示，那么这个场就是一个矢量场。对于三维空间中的场，每个点对应一个矢量，它由 X 轴、Y 轴、Z 轴上的三个分量组成。因此需要 3 个三维数组来表示矢量场，这些数组分别表示矢量在三个轴上的分量。下面以洛伦茨吸引子为例，介绍对矢量场进行可视化的一些基本方法。



mlab_vector_field.py
可视化洛伦茨吸引子的速度场

下面的代码根据洛伦茨吸引子的公式计算出 X 轴、Y 轴、Z 轴 3 个方向上的速度分量 u、v、w：

```
>>> p, r, b = (10.0, 28.0, 3.0)
>>> x, y, z = np.mgrid[-17:20:20j, -21:28:20j, 0:48:20j]
>>> u, v, w = p*(y-x), x*(r-z)-y, x*y-b*z
```

这 3 个速度分量构成一个矢量场，下面调用 quiver3d()将每个点的速度矢量用一个箭头来表示：

```
>>> vectors = mlab.quiver3d(x, y, z, u, v, w)
```

效果如图 10-15(左)所示(见文前彩插)。由于矢量场数据的网格过密，我们看不清楚矢量场的内部结构。此时可以用下面的语句，修改“Vectors”对象的一些属性以减少箭头的数量并增加箭头的长度，效果如图 10-15(右)所示(见文前彩插)。

```
>>> vectors.glyph.mask_input_points = True  # 开启使用部分数据的选项
>>> vectors.glyph.mask_points.on_ratio = 20 # 随机选择原始数据中的 1/20 个点进行描绘
>>> vectors.glyph.glyph.scale_factor = 5.0 # 设置箭头的缩放比例
```

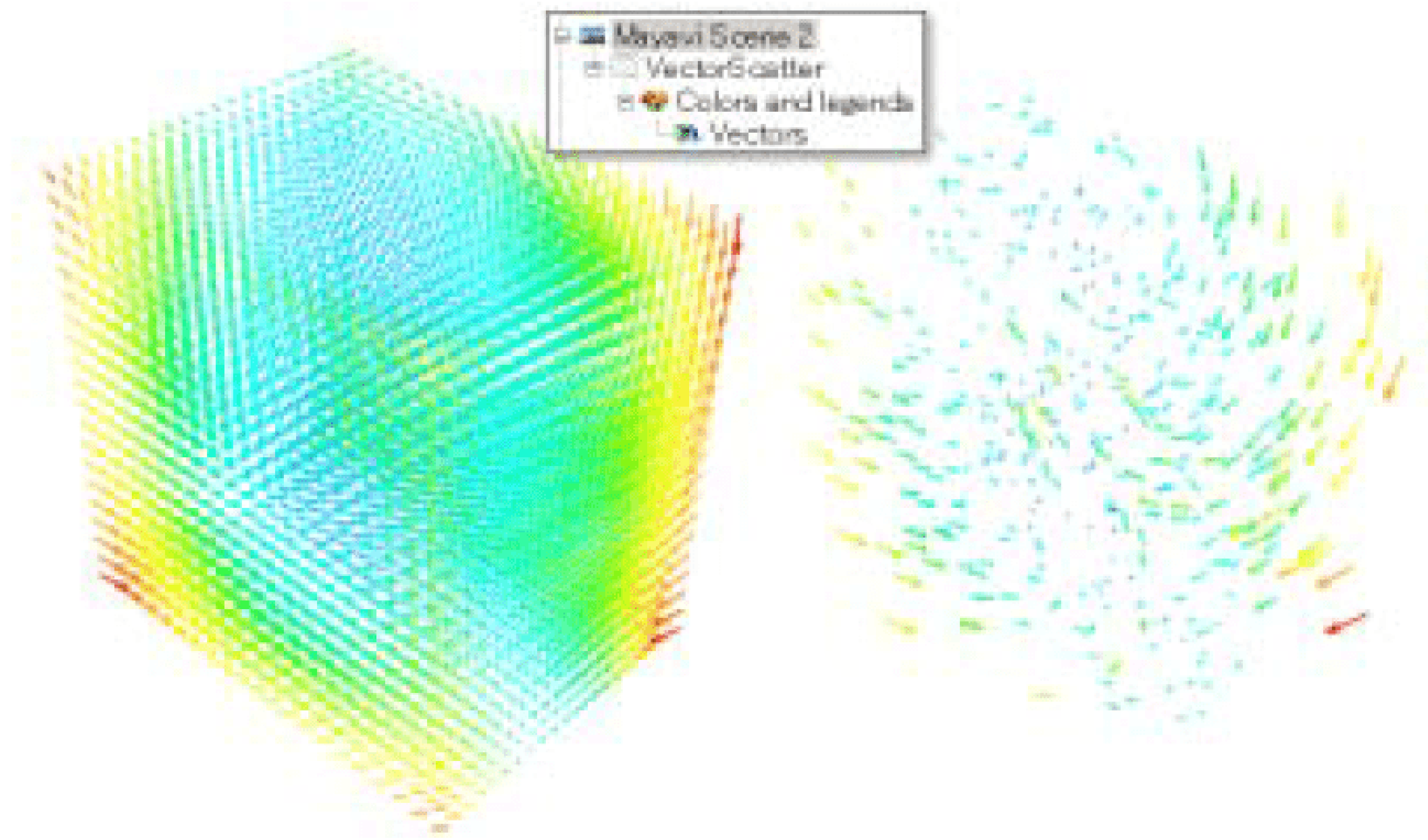


图 10-15　用矢量箭头可视化矢量场

和标量场的切面工具一样，我们也可以对矢量场进行切面显示，这样可以观察矢量场在某个切面上的形态：

```
>>> src = mlab.pipeline.vector_field(x, y, z, u, v, w)
>>> mlab.pipeline.vector_cut_plane(src, mask_points=2, scale_factor=3)
```

还可以通过矢量场计算标量场。下面通过 extract_vector_norm()在流水线中添加一个“ExtractVectorNorm”对象，它将每个点所对应矢量的长度设置为此点的标量值：

```
>>> magnitude = mlab.pipeline.extract_vector_norm(src)
```

于是可以对 magnitude 表示的标量场绘制等值面：

```
>>> surface = mlab.pipeline.iso_surface(magnitude)
>>> surface.actor.property.opacity = 0.3
```

图 10-16(左)是矢量切面工具和等值面的显示效果(见文前彩插)。下面的语句分别获取 magnitude 输出的 ImageData 对象的标量数组和矢量数组：

```
>>> magnitude.outputs[0].point_data.scalars
[58.8557548523, ..., 50.0399856567], length = 8000
>>> magnitude.outputs[0].point_data.vectors
[(0.0, -58.0, 10.0), ..., (0.0, 50.0, -2.0)], length = 8000
```

最后，还可以使用 flow()观察洛伦茨吸引子的轨迹。这相当于绘制矢量场的场线(流线)，空间中每点对应的矢量等于经过此点的场线的切线方向：

```
>>> mlab.flow(x, y, z, u, v, w)
```

图 10-16(右)是使用 flow()绘制的洛伦茨吸引子轨迹(见文前彩插)。以图中球体上的每个点为初始点计算它们对应的场线轨迹。流水线中的“Streamline”对象有许多配置选项，请读者自行进行研究。

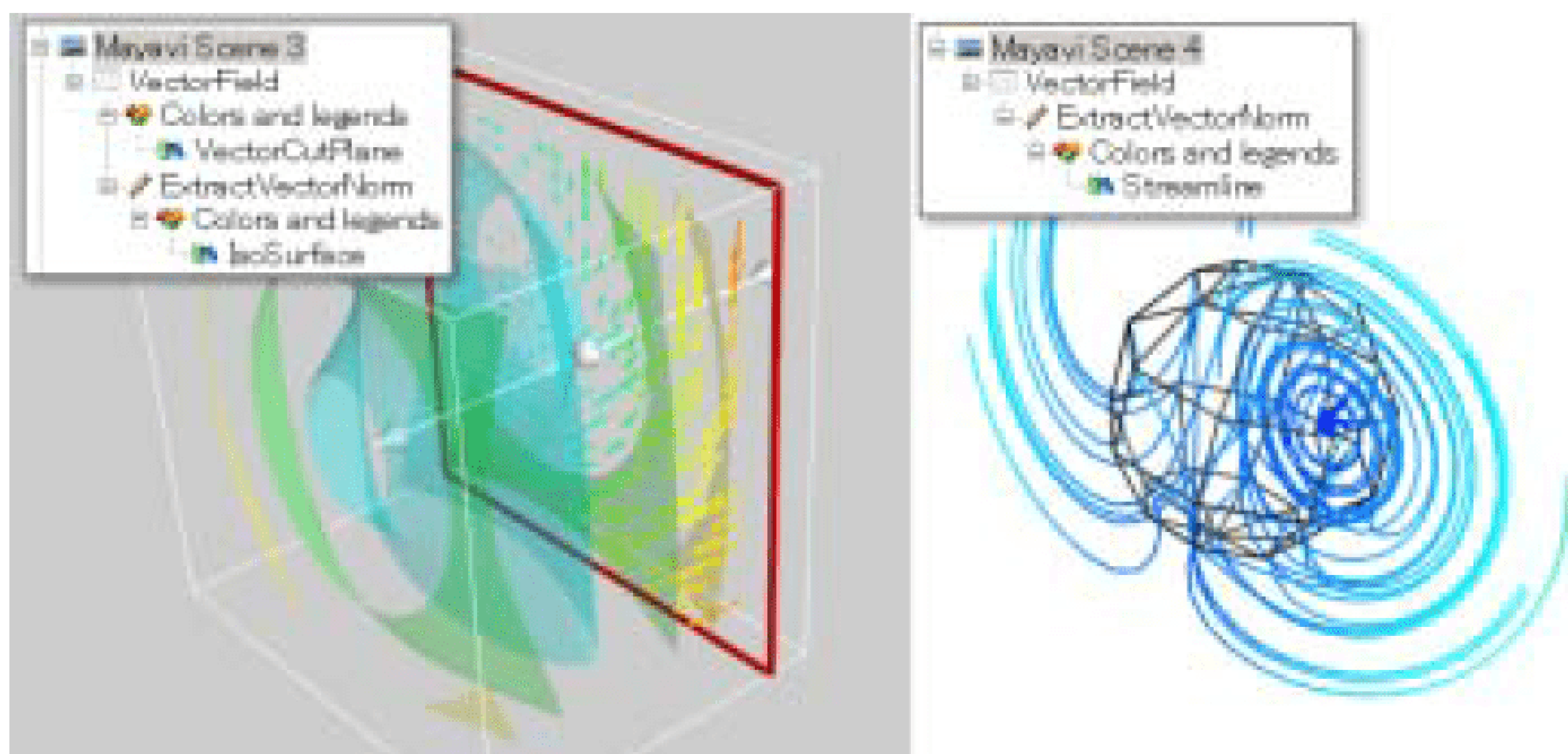


图 10-16　用矢量切面和等值面可视化矢量场(左)、用 flow()观察轨迹(右)

10.2　Mayavi 和 TVTK 之间的关系

Mayavi 建立在 TVTK 基础之上，它对 TVTK 进行了高度封装。在实际使用 Mayavi 时，我们很少需要直接和 TVTK 打交道。但是如果读者有兴趣分析 Mayavi 的源代码，为其添加新的模块，那么本节的内容可以作为一个起点。为了更好地掌握 Mayavi，本节将更深入地研究 Mayavi 和 TVTK 之间的关系。

10.2.1　显示 TVTK 流水线

在前面的介绍中，我们所看到的流水线是 Mayavi 的流水线，实际上在其内部还创建了一条 TVTK 的流水线以生成最终的场景。下面的程序使用 TVTK 流水线浏览器显示 mlab.surf()所创建的流水线。


mayavi_tvtk_pipeline.py
显示 Mayavi 创建的 TVTK 流水线

```
import numpy as np
from enthought.mayavi import mlab

x, y = np.ogrid[-2:2:20j, -2:2:20j]
z = x * np.exp( - x**2 - y**2)

face = mlab.surf(x, y, z, warp_scale=2)
mlab.axes(xlabel='x', ylabel='y', zlabel='z')
mlab.outline(face)

from enthought.tvtk.pipeline.browser import PipelineBrowser ❶
b = PipelineBrowser()
b.root_object = [mlab.gcf().scene.render_window] ❷
b.show() ❸
mlab.show()
mscene = mlab.gcf()
tscene = mscene.scene
rw = tscene.render_window
a1 = rw.renderers[0].view_props[0]
```

❶载入 TVTK 流水线浏览器，并创建一个浏览器对象。❷将 TVTK 流水线的终点(一个 RenderWindow 对象)传递给流水线浏览器的 root_object 属性，注意需要使用列表将其封装起来。❸调用流水线浏览器的 show()方法显示 TVTK 流水线。图 10-17 显示了 mlab.surf()所创建的 Mayavi 流水线和 TVTK 流水线。可以看出：TVTK 流水线十分复杂，而 Mayavi 流水线则简单许多。

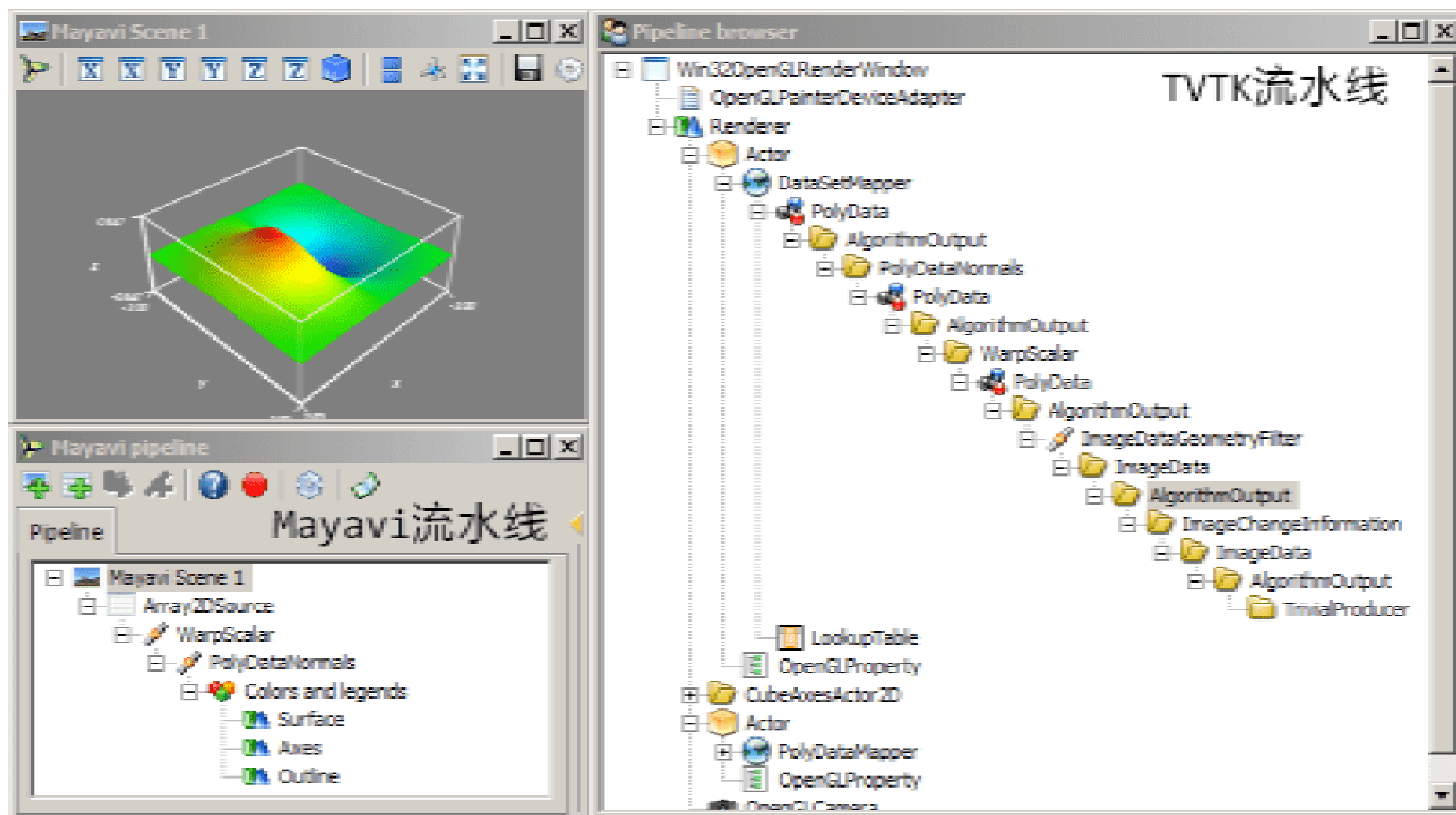


图 10-17 surf()创建的 Mayavi 流水线和 TVTK 流水线

10.2.2 两条流水线之间的关系

在 Mayavi 流水线中，根节点是一个 Mayavi 场景，可以通过 mlab.gcf()获得当前的场景，例如：

```
>>> run mayavi_tvtk_pipeline.py
>>> mscene = mlab.gcf()
>>> mscene
>>> <enthought.mayavi.core.scene.Scene object at 0x097DE630>
>>> mscene.name # 流水线中根节点的名称
'Mayavi Scene 1'
```

Mayavi 场景的 scene 属性是一个 TVTK 场景对象：

```
>>> tscene = mscene.scene
>>> tscene
>>> <enthought.mayavi.core.ui.mayavi_scene.MayaviScene object at 0x0983B030>
```

请不要被它的类名 MayaviScene 所迷惑，MayaviScene 实际上是从 TVTKScene 继承的：

```
>>> tscene.__class__.mro()
[<class 'enthought.mayavi.core.ui.mayavi_scene.MayaviScene'>,
 <class 'enthought.tvtk.pyface.ui.wx.decorated_scene.DecoratedScene'>,
 <class 'enthought.tvtk.pyface.ui.wx.scene.Scene'>,
 <class 'enthought.tvtk.pyface.tvtk_scene.TVTKScene'>,
 [[省略]]
```

TVTKScene 对 TVTK 中的 RenderWindow 对象进行了封装，可以通过其 render_window 属性获得 RenderWindow 对象。TVTK 流水线以 RenderWindow 对象为终点，因此它是 TVTK 流水线中的根节点：

```
>>> tscene.render_window.__class__
<class 'tvtk_classes.win32_open_gl_render_window.Win32OpenGLRenderWindow'>
>>> tscene.render_window.__class__.mro()
[<class 'tvtk_classes.win32_open_gl_render_window.Win32OpenGLRenderWindow'>,
 <class 'tvtk_classes.open_gl_render_window.OpenGLRenderWindow'>,
 <class 'tvtk_classes.render_window.RenderWindow'>,
[[省略]]
```

图 10-18 显示了 Mayavi 流水线和 TVTK 流水线之间的关系，请读者参照此图理解接下来所介绍的内容。

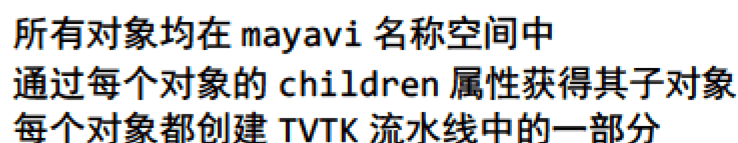


图 10-18　Mayavi 流水线和 TVTK 流水线之间的关系

在 Mayavi 流水线中，每个对象都通过 children 属性获得其子对象的列表。因此，获得 Mayavi 流水线中的某个对象是非常容易的：

```
>>> mscene.children[0]
<enthought.mayavi.sources.array_source.ArraySource object at 0x082E4D50>
>>> mscene.children[0].children[0]
<enthought.mayavi.filters.warp_scalar.WarpScalar object at 0x082F5240>
>>> mscene.children[0].children[0].children[0]
<enthought.mayavi.filters.poly_data_normals.PolyDataNormals object at 0x084B9BD0>
```

而 TVTK 流水线就比较麻烦了，每个对象都需要通过不同的属性获得其输入对象。例如下面的代码从 RenderWindow 开始，沿着流水线依次访问输入对象。其中，renderer 和 view_props 属性获得的是输入对象的列表，而 mapper、input、producer_port 和 producer 等属性则直接获得输入对象：

```
>>> rw = tscene.render_window
>>> rw.renderers[0]
<tvtk_classes.renderer.Renderer object at 0x07E99EA0>
>>> rw.renderers[0].view_props[0]
<tvtk_classes.actor.Actor object at 0x08557D50>
>>> rw.renderers[0].view_props[0].mapper
<tvtk_classes.data_set_mapper.DataSetMapper object at 0x08557F90>
>>> rw.renderers[0].view_props[0].mapper.input
```

```
<tvtk_classes.poly_data.PolyData object at 0x08557450>
>>> rw.renderers[0].view_props[0].mapper.input.producer_port.producer
<tvtk_classes.poly_data_normals.PolyDataNormals object at 0x085572A0>
```

TVTK 流水线中的每个对象都由 Mayavi 流水线中的对象产生，例如下面的代码可获得 Mayavi 流水线中的 PolyDataNormals 对象，并显示其 filter 和 outputs[0]属性，可以看到它们就是 TVTK 流水线中的对象：

```
>>> normals = mscene.children[0].children[0].children[0]
>>> normals
<enthought.mayavi.filters.poly_data_normals.PolyDataNormals object at 0x084B9BD0>
>>> normals.filter
<tvtk_classes.poly_data_normals.PolyDataNormals object at 0x085572A0>
>>> normals.outputs[0]
<tvtk_classes.poly_data.PolyData object at 0x08557450>
```

10.3 Mayavi 应用程序

Mayavi 还可以作为一个独立的应用程序使用。它的启动代码保存在“C:\Python26\Scripts”下，如果在 PATH 环境变量中添加了此路径，就可以直接在命令行中输入“mayavi2”或“mayavi2-script”来启动 Mayavi。也可以通过 Python(x,y)的开始界面或者 Windows 的“开始”菜单启动 Mayavi。Mayavi 的界面如图 10-19 所示。

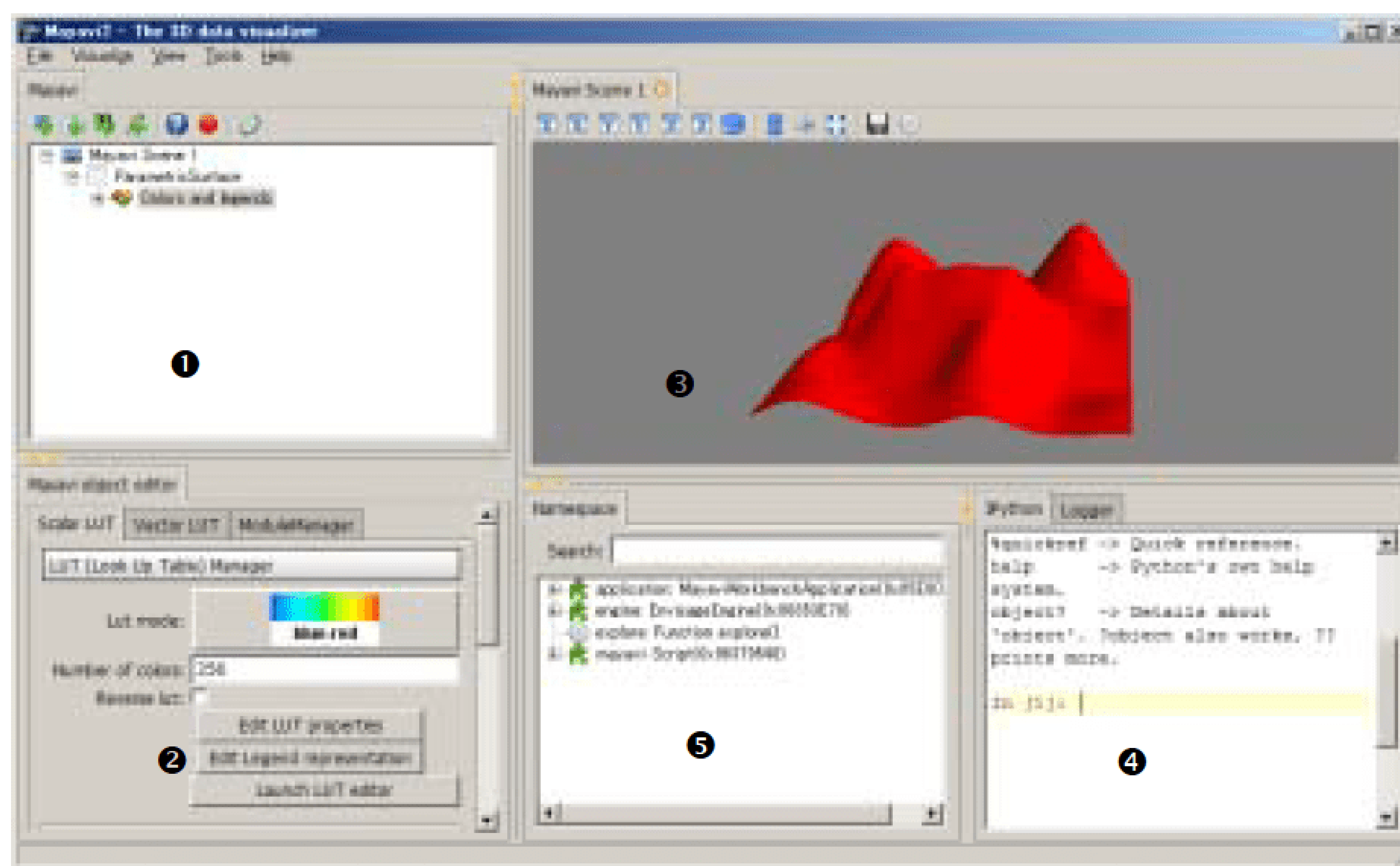


图 10-19 Mayavi 的界面截图

在此界面中，用标签页的形式显示了多个不同功能的窗格，可以通过拖动标签页和分隔栏来修改各个窗格的位置和大小。界面中各个窗格的功能如下：

❶Mayavi 流水线浏览器：管理多个场景的流水线。通过它的工具菜单可以为流水线添加各种对象，从左到右的工具按钮依次对应：场景、数据源、可视化模块和处理模块。❷Mayavi 对象编辑器：显示流水线浏览器中被选中对象的配置界面。❸场景窗口：流水线中的每个场景对象都对应一个场景窗口。❹Python 命令行：通过命令行可以输入程序命令，对 Mayavi 的流水线和各个对象进行操作，图中使用 IPython 作为命令行工具。❺对象浏览器：浏览在 Python 命令行中能够访问的所有对象。此窗格在默认界面中是隐藏的，为了显示它，可以通过菜单 “View” → “Others” 打开 “Show View” 对话框，选择 “Namespace” 后单击 OK 按钮。

在 Mayavi 的默认界面中使用标准的 Python 命令行，我们可以通过下面的步骤将 Mayavi 的命令行工具改为 IPython：

(1) 首先运行 Mayavi，并选择菜单 “Tools” → “Preferences”，在 Preferences 对话框的左侧选择 Mayavi，并在右侧的 “Genernal settings” 选项区域中选中 “Use ipython” 。

(2) 重启 Mayavi，查看控制台的标签名是否是 “IPython” 。如果不是，就进行步骤(3)。

(3) 退出 Mayavi，在命令行中输入下面的命令，安装 twisted 和 zope.interface 库。如果 twisted 安装失败，也可以去 Twisted 的官方网站下载 Windows 的安装程序进行安装。

```
easy_install twisted
easy_install zope.interface
```



http://twistedmatrix.com
Twisted 的官方网站

10.3.1 操作流水线

通过流水线浏览器的工具栏可以为流水线添加场景、数据源、可视化模块和处理模块。也可以通过流水线中对象的右键菜单为其添加子对象。Mayavi 的流水线由下列 4 种对象组成：

- 场景(Scene)：在流水线浏览器中，每个根节点都与一个场景对象相对应。
- 数据源(Data Source)：场景的子节点为数据源节点。可以通过数据源工具按钮、Mayavi 对象编辑器或场景节点的右键菜单来添加。
- 处理器(Filter)：对数据源或处理器的输出进行处理，产生新的处理结果。
- 显示组件(Module)：将数据源或处理器的输出转换成场景中的物体。

在场景中可以包含多个数据源，每个数据源经过处理器进行数据处理之后，由显示组件在场景中显示出最终的可视化结果。

下面通过一个实例介绍 Mayavi 界面的使用方法。我们将对下面两个数据文件进行可视化处理：



data\fire_ug.vtu, data\room_vis.wrl
要进行可视化处理的数据文件

其中，“fire_ug.vtu”是一个 VTK 的 XML 格式的文件，其中保存一个 UnstructuredGrid 数据集，它表示当房间中的一角着火时，房间中各点的温度以及空气的流动速度。“room_vis.wrl”文件是房间的内部构造以及着火点的 3D 模型。

UnstructuredGrid 数据集是最一般性的数据集，它的每个点的坐标以及点和单元之间的关系都必须显式指定。后面我们将使用 Mayavi 作为工具深入研究 UnstructuredGrid 数据集的构造。现在，首先使用 Mayavi 对这两个数据文件进行可视化处理：

(1) 开启一个新的 Mayavi2 应用程序。选择菜单“File”→“Load data”→“Open file”，在对话框中打开“room_vis.wrl”文件。由于此文件中保存的是房间的立体模型，因此不需要任何显示组件，即可在场景中显示出来。

(2) 再次使用“Open file”菜单打开“fire_ug.vtu”文件。我们需要为其添加显示组件才能进行可视化，因此这时场景不会有任何变化。

(3) 当在流水线中选中“fire_ug.vtu”的数据源节点时，Mayavi 对象编辑器(名为“Mayavi object editor”的窗格)中将显示数据源的各种选项，这里可以选择点的标量数据名和矢量数据名。我们使用默认设置，即标量数据名为 t、矢量数据名为 uvw。t 表示温度，uvw 表示速度矢量。而标量数据还可以有 u、v、w 和 mfrac 等几种选择。其中，u、v 和 w 为速度在三个坐标轴上的分量，它们都是标量值。

(4) 我们首先为“fire_ug.vtu”数据源绘制一个外框。选中数据源节点，然后选择菜单“Visualize”→“Modules”→“Outline”。这时在数据源节点下将创建一个名为“Colors and legends”的颜色和图示配置节点，其下有一个名为“Outline”的显示组件。到目前为止，流水线和场景如图 10-20 所示。

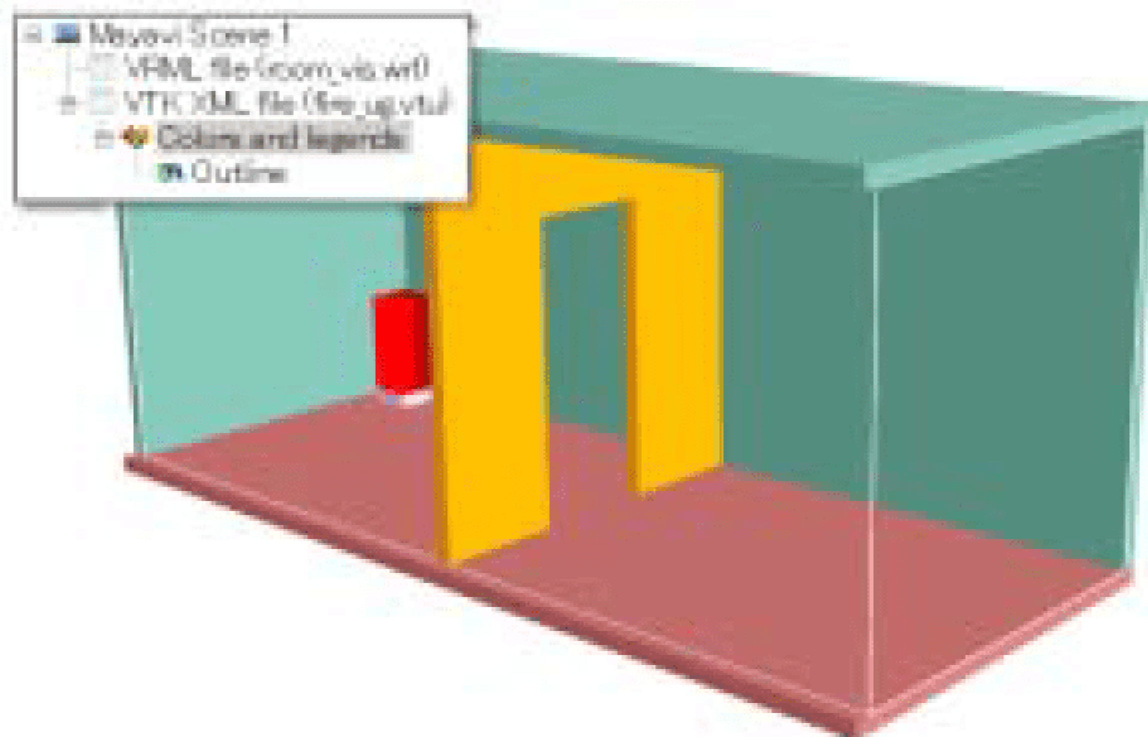


图 10-20　房间模型和数据集的外框

下面用前面介绍过的切面、等值面及流线对数据集进行可视化处理。

(1) 使用组件菜单添加两个组件：“ScalarCutPlane”和“IsoSurface”。

(2) 在流水线中选中“IsoSurface”节点，并在 Mayavi 对象编辑器中修改其属性。首先在“Contours”选项卡中，选中“Auto contours”选项，并且设置“Number of countours”为 15。然后在“Actor”选项卡中将“Opacity”设置为 0.1。这样就在温度的取值范围内自动构造了 15 个半透明的等值面。

(3) 可以在场景中直接修改切面的位置和方向，也可以在流水线中选中“ScalarCutPlane”节点，然后在 Mayavi 对象编辑器中修改其属性。

(4) 选中“Colors and legends”节点，并且选中“Show legend”选项。场景中将出现一个颜色条，它表示颜色和温度之间的对应关系。在场景中可以通过鼠标调整颜色条的方向、大小和位置。此时的流水线和场景如图 10-21 所示(见文前彩插)。

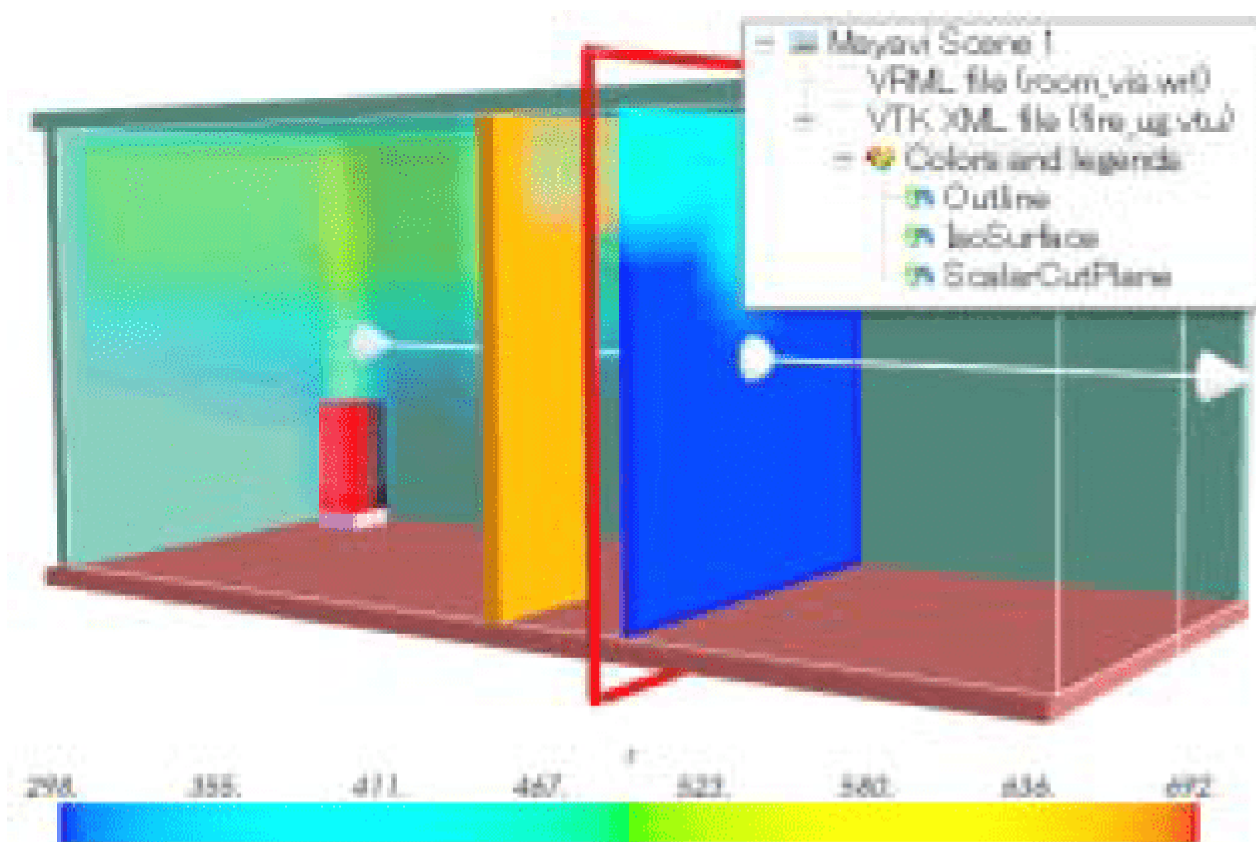


图 10-21　使用切面和等值面观察温度分布

(5) 选中“fire_ug.vtu”数据源节点，在对象编辑器中通过“Point scalars name”修改与每个点对应的标量数据名。可以看到场景中的颜色条、切面及等值面都会即时刷新，对选中的标量数据进行可视化。最后请选择温度标量 t。

(6) 使用组件菜单添加“Streamline”组件(流线组件)，它可以对表示速度的矢量数据进行可视化。场景中将生成一个作为流线源的球体，以及从球体上各点出发的流线轨迹。流线上各点的颜色用其所在位置的标量值对应的颜色表示。通过 Mayavi 对象编辑器可以修改流线源以及流线的形状、大小等属性。

(7) 数据集中没有表示速率(速度的大小)的标量数据。我们可以通过添加处理器从速度矢量数据计算出速率。选中“fire_ug.vtu”数据源节点，然后选择菜单“Visualize”→“Filters”→“Extract Vector Norm”，为数据集添加一个名为“ExtractVectorNorm”的处理器。

(8) 可以为“ExtractVectorNorm”节点添加切面和等值面等显示组件，也可以将现有的显示组件移动到它下面。用鼠标左键拖放“Colors and legends”节点到“ExtractVectorNorm”节点之上。这样一来，所有的显示组件都会对“ExtractVectorNorm”处理器的输出数据进行可视化，于是我们得到了速率的可视化结果，效果如图 10-22 所示(见文前彩插)。

(9) 为了还原成温度的可视化结果，只需要将“Colors and legends”节点拖放到“fire_ug.vtu”

数据源节点之上即可。由于“ExtractVectorNorm”处理器不改变数据集的矢量数据，因此流线的形状不会发生变化，而流线的颜色会随着标量数据的变化而改变。

(10) 最后，可以使用菜单“File”→“Save Visualization”将当前的可视化状态保存成文件。使用菜单“File”→“Load Visualization”则可以从文件恢复以前保存的可视化状态。

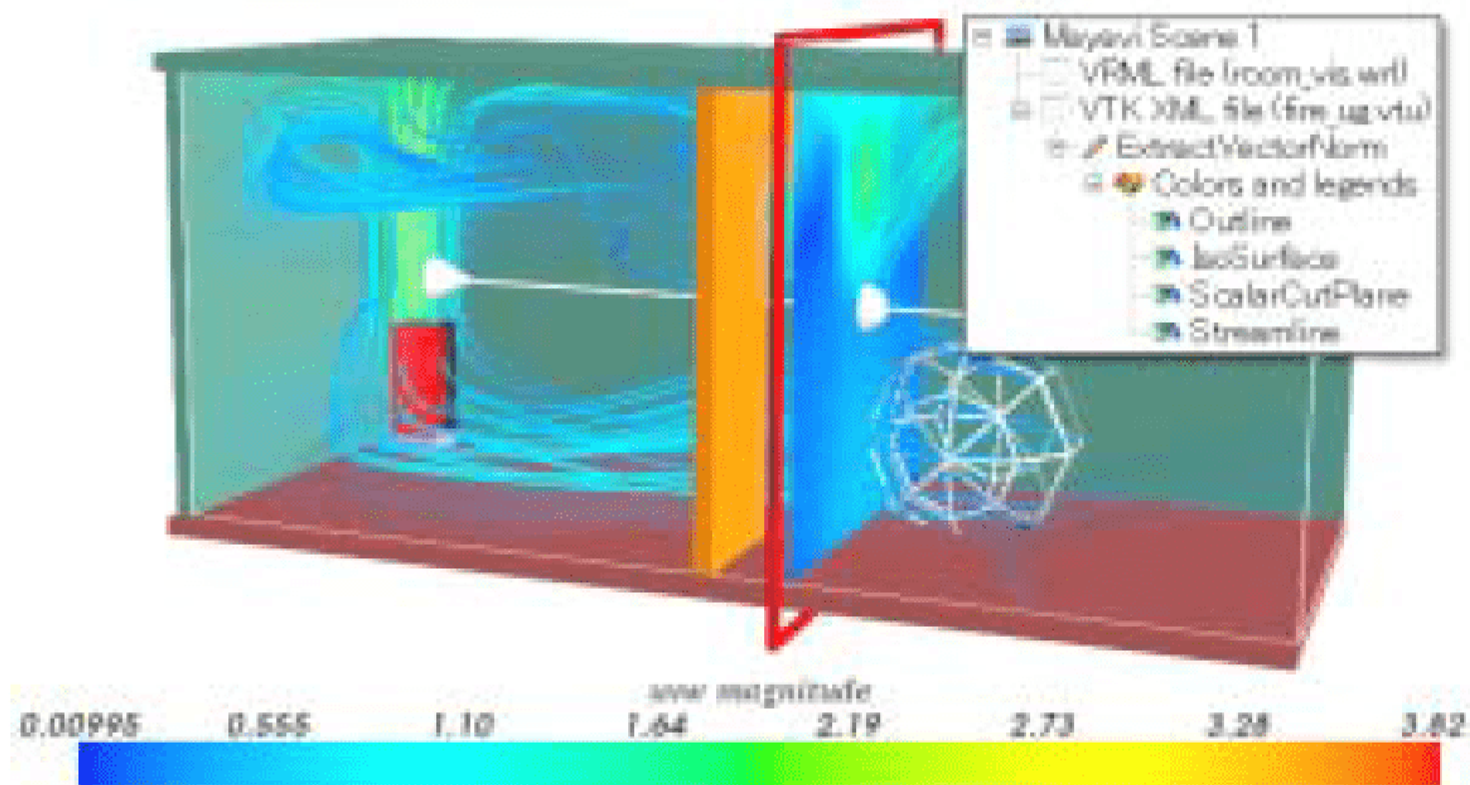


图 10-22　使用切面、等值面及流线观察速率和速度的分布情况

10.3.2　命令行和对象浏览器

通过命令行和对象浏览器，可以很方便地查看和修改 Mayavi 的各个对象的属性，了解对象的内部结构。下面我们用这两个工具查看“fire_ug.vtu”数据源输出的 UnstructuredGrid 数据集的内部构造。



本节涉及数据集的内部构造，为了便于理解，请读者首先仔细阅读并理解前面 TVTK 章节中数据集的相关内容。

首先打开对象浏览器窗格，其中显示了 4 个对象：

- application:MayaviWorkbenchApplication：表示当前应用程序的对象。
- engine:EnvisageEngine：在最高层管理 Mayavi 的各种对象。
- explore:Function：调用 explore(x)将打开一个浏览对象 x 各种属性的窗格。
- mayavi:Script：通过它可以调用 engine 中的一些常用方法。

对象浏览器中显示的都是命令行名称空间中的对象，因此在命令行窗格输入“engine?”和“mayavi?”可以查看它们的相关说明。输入“explore(application)”，将打开一个新的窗格以浏览 application 对象的内容。

如图10-23 所示，在对象浏览器窗格中一层一层地展开engine对象的属性，直到找到VTKXMLFileReader 对象，它就是流水线中的“fire_ug.vtu”数据源节点。



图 10-23　展开 engine 对象的属性，直到找到 VTKXMLFileReader 对象

进一步找到 VTKXMLFileReader 对象的子节点“.outputs:List(1)”，它是一个长度为 1 的列表，它的第 0 个元素就是我们要进行分析的 UnstructuredGrid 对象。为了方便观察它，在命令行中输入：

```
>>> data = engine.current_scene.children[1].outputs[0]
```

命令行的名称空间中多了一个 data 变量以表示 UnstructuredGrid 对象，因此对象浏览器中也同时多了一个名为“data:UnstructuredGrid”的根节点。读者可以使用对象浏览器研究一下 data 对象的各个属性，为了表述方便，下面我们使用命令行查看它的各种属性：

```
>>> data.points # 保存所有点的坐标
[(0.0, 0.0, 0.0), ..., (6.0, 2.5, 2.3999998569488525)], length = 12323
>>> data.point_data.number_of_arrays # 每个点对应有 6 个数据
6
```

下面的语句获得与每个点对应的 6 个数据的名称，以及默认的标量数据和矢量数据的名称：

```
>>> [data.point_data.get_array_name(i) for i in xrange(6)]
['t', 'uvw', 'u', 'v', 'w', 'mfrac']
>>> data.point_data.scalars.name
't'
>>> data.point_data.vectors.name
'uvw'
```

下面查看单元和点之间的关系：

```
>>> data.get_cells().number_of_cells # 获得单元个数
10462
```

```
>>> data.get_cells().data # 用来指定单元和点之间关系的数组
[8.0, ..., 12321.0], length = 94158
>>> data.get_cell(0) # 获得第 0 个单元，它是一个六面体
<tvtk_classes.hexahedron.Hexahedron object at 0x08CF3120>
>>> data.get_cell(0).point_ids # 获得构成第 0 个单元的点的下标
[0, 1, 721, 720, 40, 41, 761, 760]
```

最后查看与单元对应的数据的个数，因为值为 0，所以数据集中没有和单元对应的数据：

```
>>> data.cell_data.number_of_arrays
0
```

通过上面的分析可以得出如下结论：

- UnstructuredGrid 对象由 12323 个点和 10462 个单元构成。
- 每个点对应 6 个数据，其中名为"uvw"的数据为矢量数据，其他均为标量数据。
- 没有与单元对应的数据。
- 点和单元之间的关系由 CellArray 显式指定。

为了观察 UnstructuredGrid 对象的单元和点之间的关系，我们使用“Extract Edges”处理器抽出每个单元的边线，然后使用“Surface”组件将其在场景中显示出来。相信读者已经学会了如何使用界面在流水线中添加这两个对象，这里为了演示命令行的用法，使用命令行进行处理：

```
>>> from enthought.mayavi.filters.api import ExtractEdges
>>> from enthought.mayavi.modules.api import Surface
>>> extract_edge = ExtractEdges()
>>> engine.add_filter(extract_edge, engine.current_scene.children[1])
>>> engine.add_module(Surface(), extract_edge)
```

上面的语句首先载入 ExtractEdges 和 Surface，然后调用 engine 对象的 add_filter()和 add_module()方法分别添加这两个对象。第一个参数为被添加的对象，第二个参数为它的上一级对象。为了突出显示，我们可以用鼠标在流水线中右击某个项目，从弹出菜单中选择“Hide/Show”，暂时关闭一些不相干的组件的显示。最后的效果如图 10-24 所示(见文前彩插)。

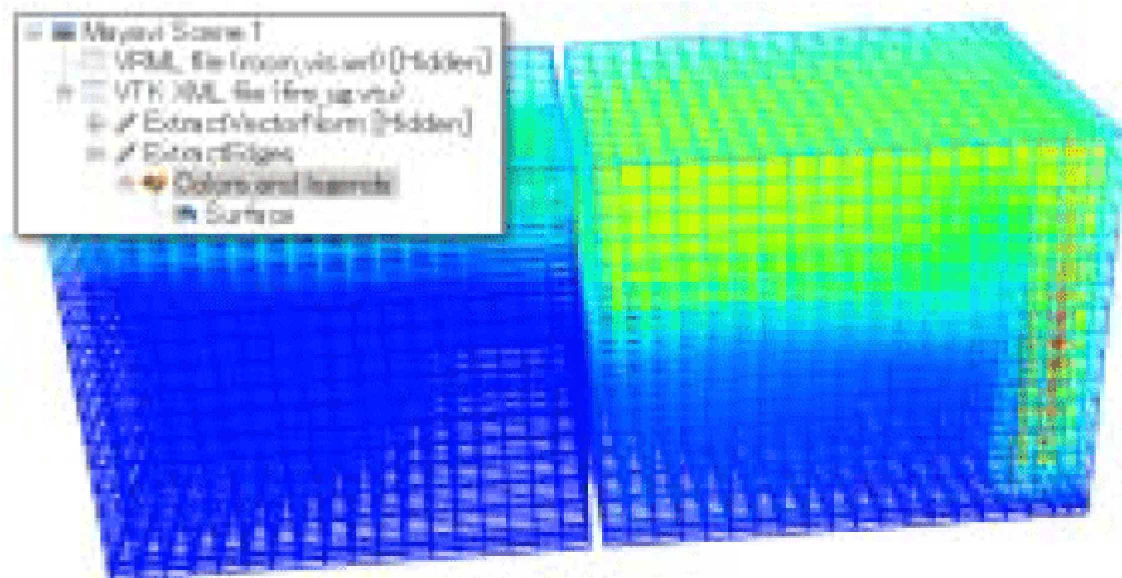
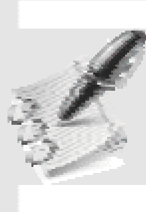


图 10-24　UnstructuredGrid 对象的单元边线

由此可以看出单元和点之间的关系是十分规则的，但是由于房间内的分隔墙所在位置不属于数据集，因此在中间有一个断层，为了表示这个断层只能使用 UnstructuredGrid 数据集显式地指定点和单元之间的关系。如果没有这个断层，那么整个空间中的数据可以使用 ImageData 数据集进行描述。

10.4 将 Mayavi 嵌入到界面中

Mayavi 除了能够单独作为应用程序使用之外，也可以通过 Trait 属性嵌入到使用 TraitsUI 制作的用户应用程序的界面中，下面的程序演示了这一过程：



mayavi_embed_demo.py
将 Mayavi 场景嵌入到 TraitsUI 界面中

```
from enthought.traits.api import HasTraits, Button, Instance
from enthought.traits.ui.api import View, Item, VGroup
from enthought.tvtk.pyface.scene_editor import SceneEditor ❶
from enthought.mayavi.tools.mlab_scene_model import MlabSceneModel
from enthought.mayavi.core.ui.mayavi_scene import MayaviScene
from enthought.mayavi import mlab

class DemoApp(HasTraits):
    plotbutton = Button(u"绘图")
    # mayavi 场景
    scene = Instance(MlabSceneModel, ()) ❷

    view = View(
        VGroup(
            # 设置 mayavi 的编辑器
            Item(name='scene',  ❸
                editor=SceneEditor(scene_class=MayaviScene),
                resizable=True,
                height=250,
                width=400
            ),
            'plotbutton',
            show_labels=False
        ),
        title=u"在 TraitsUI 中嵌入 Mayavi"
    )

    def _plotbutton_fired(self):
        self.plot()

    def plot(self):
        mlab.test_mesh() ❹

app = DemoApp()
app.configure_traits()
```

❶除了从 Traits 和 Traits.UI 模块导入之外，还分别从不同的模块导入了 SceneEditor、MlabSceneModel 和 MayaviScene 这 3 个类。

❷MlabSceneModel 表示 Mayavi 的场景模型，它在 MVC 模式中属于模型(Model)对象，因此程序中用它定义模型类 DemoApp 的 Trait 属性 scene。

❸scene 属性在界面中将呈现为一个 Mayavi 的三维场景，因此在视图的 Item 定义中，用 editor 参数指定一个编辑器，让它能正确显示 scene 所代表的模型。SceneEditor 是用来创建场景编辑器的工厂类，通过其 scene_class 参数指定真正创建场景对象的类 MayaviScene。

❹程序中还创建了一个 plotbutton 按钮，当此按钮被单击时，调用_plotbutton_fired()，最终会调用绘制场景的 plot()方法。在 plot()方法中，调用 mlab 模块的 test_mesh()，在场景中创建一个如图 10-25 所示的很酷的曲面体。

下面看一个有些实用价值的程序。用户输入一个由 x、y、z 变量构成的表达式，例如“x*x+y*y+z*z”。程序使用此表达式计算指定范围内的三维标量场，并添加等值面和切面工具对标量场进行可视化。等值面的数值可以自动计算，也可以通过界面上的滚动条进行配置；而切面的位置和方向则可以直接在场景中用鼠标进行操作。由于程序较长，这里仅列出和绘制三维场景有关的代码。程序的界面如图 10-26 所示。



mayavi_embed_fieldviewer.py
用 TraitsUI 和 Mayavi 编写的三维标量场观察器

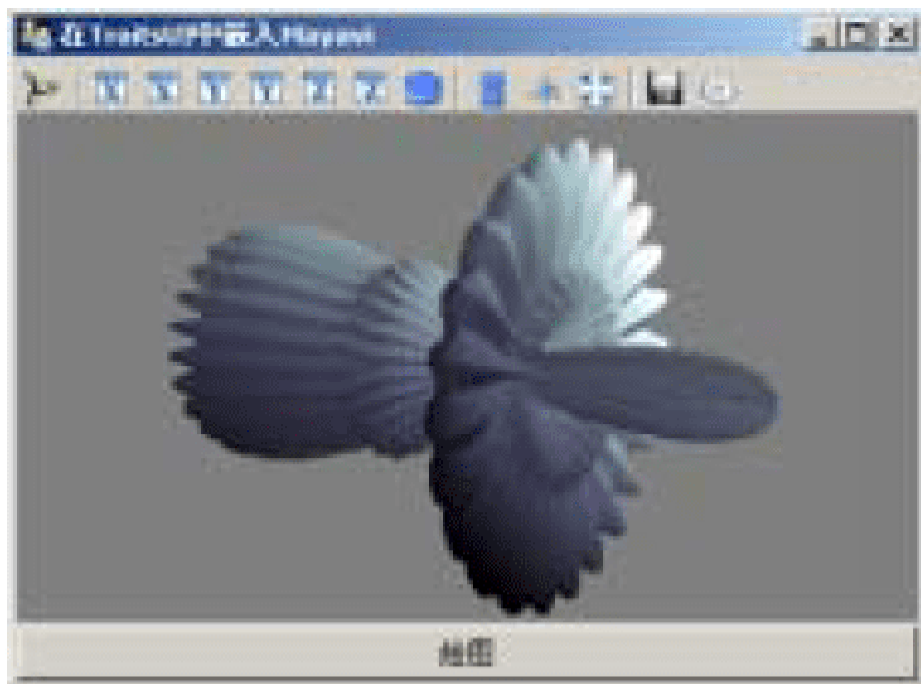


图 10-25　将 Mayavi 嵌入到用 TraitsUI 制作的界面中

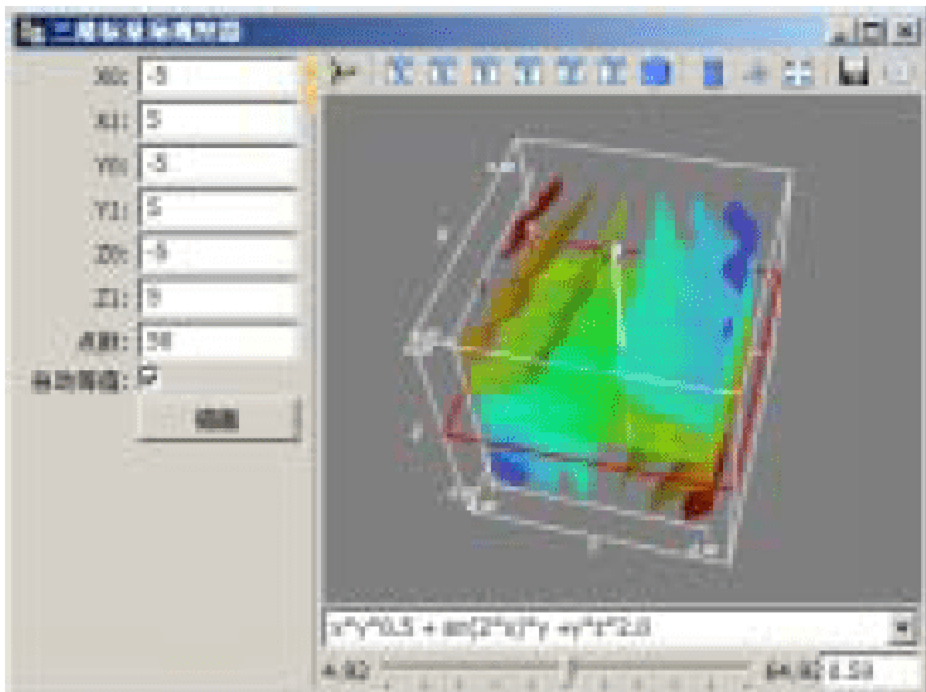


图 10-26　三维标量场观察器：x*y*0.5+2*y*sin(2*x)+y*z*2.0

```
def plot(self):
    "绘制场景"
    # 产生三维网格
    x, y, z = np.mgrid[ ❶
        self.x0:self.x1:1j*self.points,
        self.y0:self.y1:1j*self.points,
        self.z0:self.z1:1j*self.points]
    # 根据函数计算标量场的值
```

```
        scalars = eval(self.function)  ❷
        mlab.clf() # 清空当前场景
        # 绘制等值平面
        g = mlab.contour3d(x, y, z, scalars, contours=8, transparent=True) ❸
        g.contour.auto_contours = self.autocontour
        mlab.axes() # 添加坐标轴
        # 添加一个 X-Y 的切面
        s = mlab.pipeline.scalar_cut_plane(g)
        cutpoint = (self.x0+self.x1)/2, (self.y0+self.y1)/2, (self.z0+self.z1)/2
        s.implicit_plane.normal = (0,0,1) # x cut
        s.implicit_plane.origin = cutpoint
        self.g = g ❹
        self.scalars = scalars
        # 计算标量场的值的范围
        self.v0 = np.min(scalars)
        self.v1 = np.max(scalars)
```

用户单击“描画”按钮之后，将调用 plot()进行绘图。❶首先计算三维标量场的网格，注意我们使用 mgrid 快速产生三维网格，其中的 x0、x1、y0、y1、z0、z1、points、function 等都是模型类的 Trait 属性，可以通过界面上的控件直接修改这些属性的值。❷由于用户输入的三元函数是一个字符串，这里用 eval()对字符串进行求值，在字符串中可以使用 x、y、z 等局域变量。

❸清空当前场景之后，调用 mlab 模块中的 contour3d()、axes()、pipeline.scalar_cut_plane()等在场景中添加等值面、坐标轴和切面。mlab 模块默认对当前场景进行处理，如果应用程序有多个场景需要分别在其中绘图时，可以通过 figure 参数指定需要进行处理的场景，例如：

```
mlab.axes(figure=self.scene.mayavi_scene)
```

其中，self.scene 是 MlabSceneModel 对象，其 mayavi_scene 属性是真正表示场景的 Scene 对象。

❹最后更新模型对象的几个属性，其中变量 g 是 contour3d()的返回值，它表示场景中的等值面。self.v0 和 self.v1 是标量场的最小值和最大值，它们设置等值面滚动条的取值范围。

当与 contour 属性相对应的滚动条控件(位于三维场景的下方)的值发生变化时，将调用下面的_contour_changed()方法修改保存等值面数值的列表：

```
def _contour_changed(self):
    "等值面的值改变事件响应"
    if hasattr(self, "g"):
        if not self.g.contour.auto_contours:
            self.g.contour.contours = [self.contour]
```

当“自动等值”复选框控件改变时，在 autocontour 属性的事件监听函数_autocontour_changed()中改变等值面对象 g 的自动等值面选项：

```
def _autocontour_changed(self):
    "自动计算等值面的设置改变事件响应"
    if hasattr(self, "g"):
        self.g.contour.auto_contours = self.autocontour
        if not self.autocontour:
            self._contour_changed()
```

第 11 章

VPython——制作3D演示动画

VPython 是一套简单易用的三维图形库，使用它可以快速创建三维场景和动画。和 TVTK 相比，它更适合于创建交互式的三维场景，而 TVTK 则更适合于对数据进行三维可视化。本章将通过几个实例介绍如何使用 VPython 制作实时、交互式的三维动画演示程序。在 VPython 的安装目录下有数十个例子，并且有很详细的文档，读者可以参照这些资料进行更深入的学习。下面是 VPython 默认安装时的路径：

```
c:\Python26\Lib\site-packages\visual\
```

11.1 场景、物体和照相机

我们从最简单的例子开始，下面的程序创建一个窗口并且在其中显示一个立方体，运行结果如图 11-1 的左图所示[①]。

```
from visual import *
box()
```

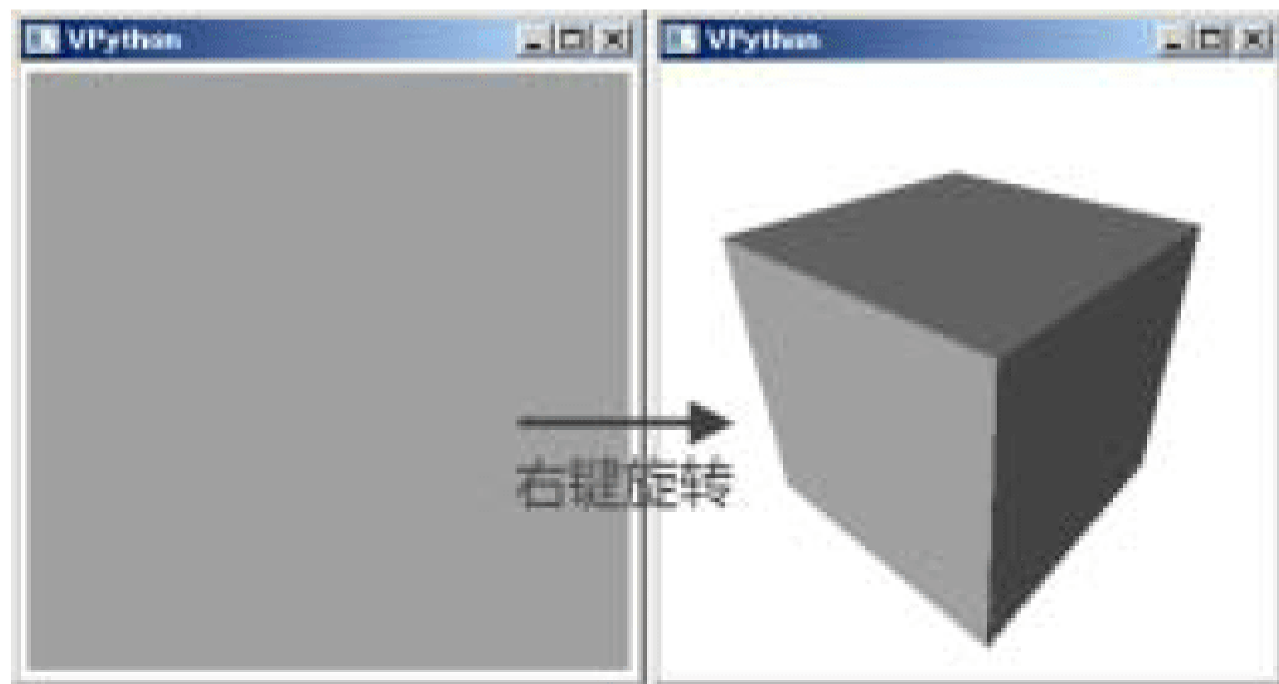


图 11-1　用 VPython 绘制的立方体：默认为俯视图(左)，旋转照相机之后(右)

程序中，首先从 visual 模块载入所有内容，然后通过 box()创建一个立方体对象，创建此

① 为了印刷方便，实际上在调用 box()之前设置了场景的背景色为白色。

立方体对象的同时将显示一个标题为“VPython”的场景窗口。由于我们没有给 box()传递任何参数，因而立方体的所有属性都将使用下面的默认值：

- 立方体的中心在三维空间中的坐标为(0, 0, 0)，即坐标原点。
- 立方体的大小为(1, 1, 1)，即长、宽、高都为 1 个单位。
- 立方体的颜色为白色。

窗口中显示的是通过一个虚拟的照相机观察三维场景时的画面。而照相机的默认位置是从 Z 轴的上方往下看(俯视图)，它自动调照相机的位置，使它正好能观察到场景中的所有物体。于是我们看到的是一个刚好充满场景窗口的正方形。在场景窗口中，同时按住鼠标左右键，并上下移动鼠标可以对场景进行缩放；按住鼠标右键移动鼠标可以对场景进行旋转。缩放和旋转场景其实都是对照相机的位置和方向进行修改，场景中物体的位置并没有发生变化，只是我们观察物体的距离和角度改变了。

VPython 可以在 IPython 中交互式地使用，启动 IPython 之后，只需要先执行：

```
>>> from visual import *
```

然后就可以在 IPython 中通过输入命令交互式地创建三维场景。需要注意的是，如果关闭了场景窗口，IPython 也随之结束。因此不要直接关闭场景窗口，而使用下面的语句将场景窗口隐藏：

```
>>> scene.visible = False
```

图 11-2 是在 IPython 中交互式地创建三维场景的一个例子，由此可知，通过 IPython 能够控制多个场景窗口。

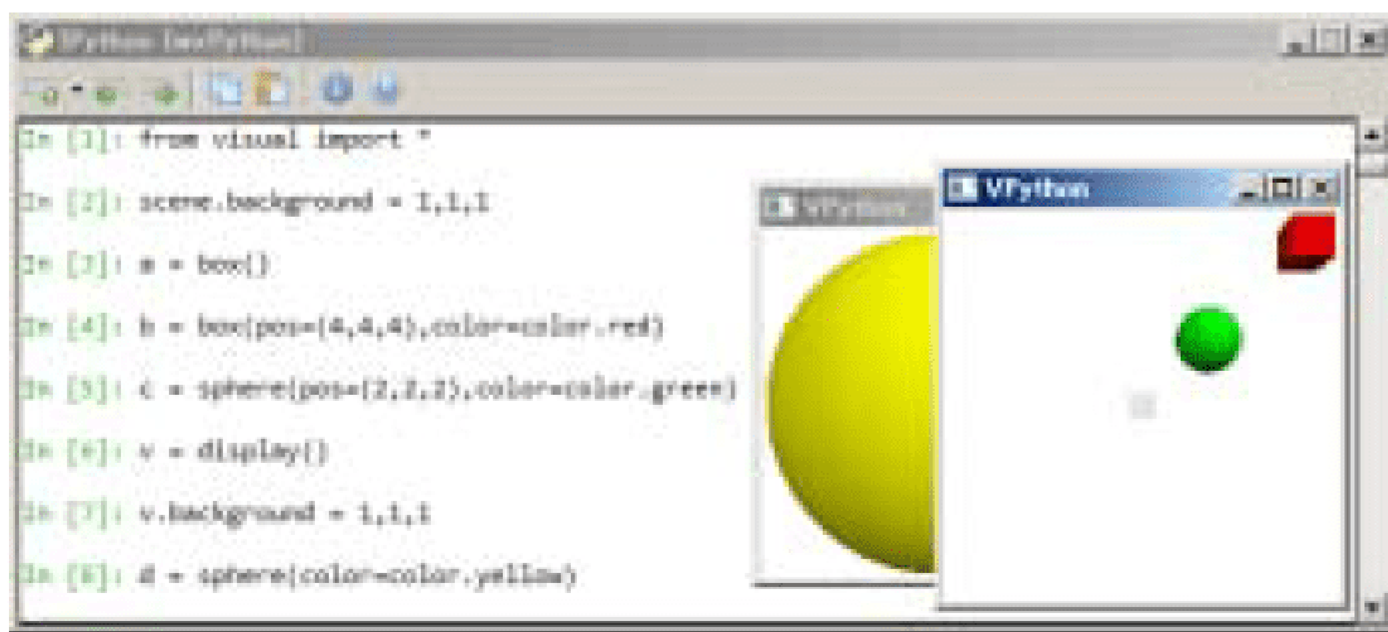


图 11-2　在 IPython 中交互式地使用 VPython

下面的程序可以帮助读者理解照相机的位置和坐标轴之间的关系，程序输出的三维场景如图 11-3 所示。



vpython_axis.py
通过绘制 X、Y、Z 坐标轴观察照相机的默认位置

```
from visual import *
display(title=u"坐标轴".encode("gb2312"), width=300, height=300, background=(1,1,1)) ❶
arrow(pos=(1,0,0), axis=(1,0,0), color=(1,0,0)) ❷
arrow(pos=(0,1,0), axis=(0,1,0), color=(0,1,0))
arrow(pos=(0,0,1), axis=(0,0,1), color=(0,0,1))
```

❶首先，调用 display()创建一个场景窗口，并且指定了窗口的标题、宽度和高度。标题必须使用 Windows 系统默认的编码标准，因此为了显示中文，需要将 Unicode 转换为 GB2312 编码。

❷调用 3 次 arrow()，创建了三个箭头物体，通过下面的关键字参数设置箭头的属性：

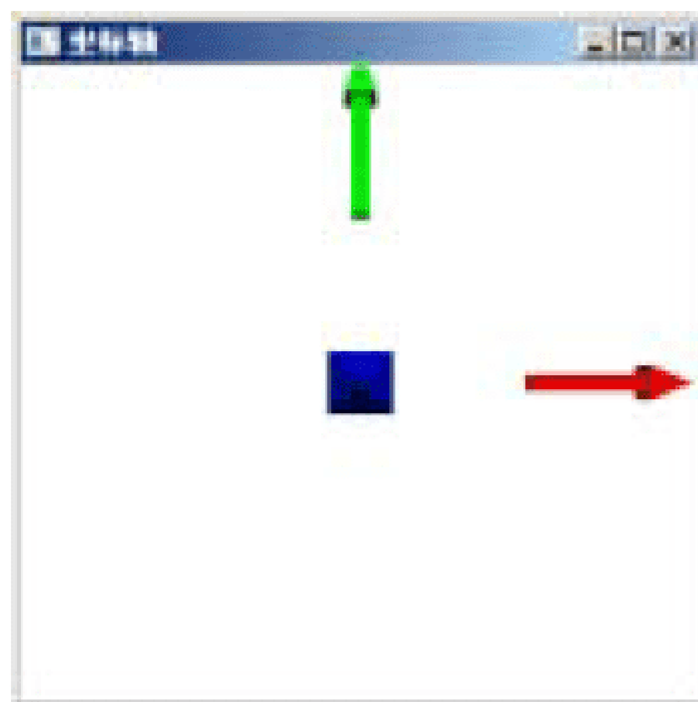


图 11-3　绘制场景的坐标轴，红、绿、蓝分别表示 X 轴、Y 轴、Z 轴

- 箭头的起点坐标通过pos 参数指定，分别为(1,0,0)、(0,1,0)、(0,0,1)，坐标用三个元素的元组表示。这三个坐标分别在 X、Y、Z 坐标轴之上。
- 箭头的方向和长度使用 axis 参数指定，其值为三维空间的矢量，矢量也用三个元素的元组表示，程序中使用的三个矢量正好是三个坐标轴的方向，长度都为 1。
- 通过 color 参数指定箭头物体的颜色，颜色也用三个元素的元组表示，每个元素的取值都在 0 到 1 之间，分别表示红、绿、蓝三种颜色的成分。

通过观察图 11-3 中三个箭头的位置(见文前彩插)，我们可以知道：窗口的中心为坐标原点，X 轴的方向为从左到右，Y 轴的方向为从下到上，Z 轴的方向为从屏幕里到屏幕外。此时的照相机位于 Z 轴正方向上的某点，沿着 Z 轴负方向俯视。

11.1.1　控制场景窗口

在 visual 模块中有一个全局变量 scene，表示默认的场景窗口对象，它同时也是初始情况下的当前窗口：

```
>>> from visual import *
>>> scene # 场景窗口的类型为display
<visual.ui.display object at 0x032BF600>
```

必须在场景中放置物体，场景窗口才会显示出来。如果用 display()创建了新的窗口对象，那么它将变成当前窗口。用 box()等函数创建的物体都将被放到当前窗口中。下面的语句调用 display()创建了一个新的窗口对象：

```
>>> scene2 = display(title='Scene2', x=0, y=0, width=600, height=200,  ...
    center=(5,0,0), background=(1,1,1))
```

执行上面的语句之后，将创建一个标题为“Scene2”的窗口，其左上角的屏幕坐标为(0,0)，

宽和高分别为 600 像素和 200 像素。通过 center 参数指定照相机正对的坐标为(5,0,0)，也就是说，窗口中心点的三维坐标为(5,0,0)。通过 background 参数指定窗口的背景色。要显示此窗口，我们需要往里面放物体：

```
>>> box(color=(0.5,0.5,0.5))
>>> <visual.primitives.box object at 0x0334F090>
>>> box(pos=(5,0,0), color=color.red)
>>> <visual.primitives.box object at 0x0334F120>
```

我们在场景中放置了两个立方体，第一个放在了默认坐标(0,0,0)处，其颜色为灰色；第二个放在了坐标(5,0,0)处，颜色为红色。红色立方体在窗口的中心，和我们设置的窗口的 center 参数一致，如图 11-4 所示(见文前彩插)。

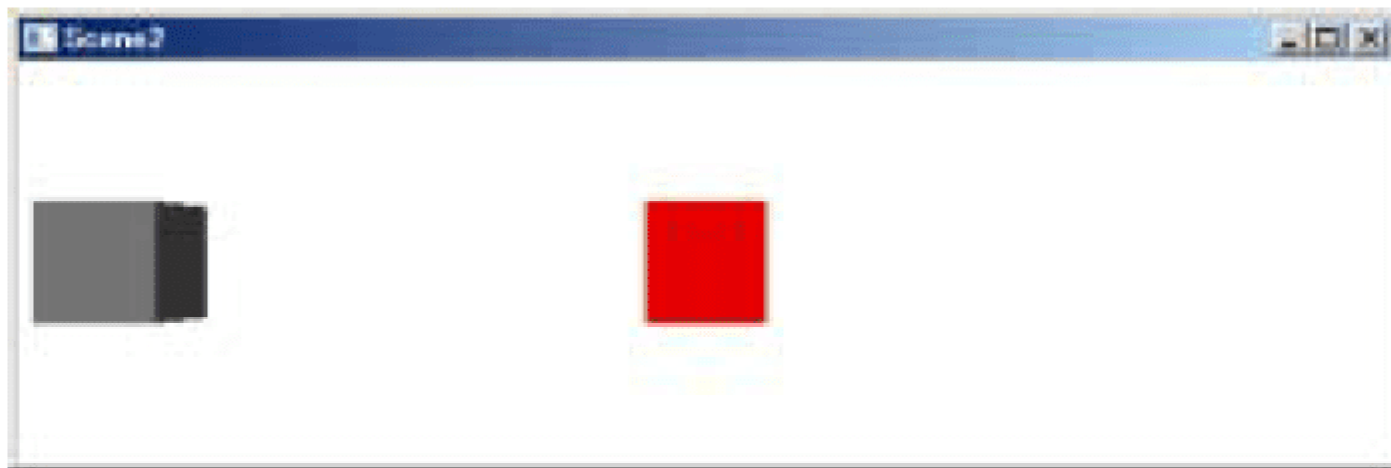


图 11-4　在场景中放置立方体

可以调用窗口对象的 select()方法使其成为当前窗口，通过 display.get_selected()可以获得当前的窗口对象：

```
>>> scene.select()
>>> scene.background=(1,1,1)
>>> sphere(color=color.yellow)
<visual.primitives.sphere object at 0x0331D7B0>
>>> scene2.select()
>>> sphere(pos=(2.5,0,0), color=color.blue)
<visual.primitives.sphere object at 0x0331D810>
>>> display.get_selected() == scene2
True
```

上面的代码先将 scene 改为当前窗口，然后在其中创建一个黄色球体。接着将 scene2 改为当前窗口，在其中创建一个蓝色球体，放在坐标(2.5,0,0)处。最后调用 display.get_selected()检查当前窗口是否是 scene2。执行这段程序之后，将出现两个场景窗口，默认窗口的标题为 VPython，其中有一个球体；我们自己创建的窗口的标题为“Scene2”，其中有两个立方体和一个球体，如图 11-5 所示(见文前彩插)。

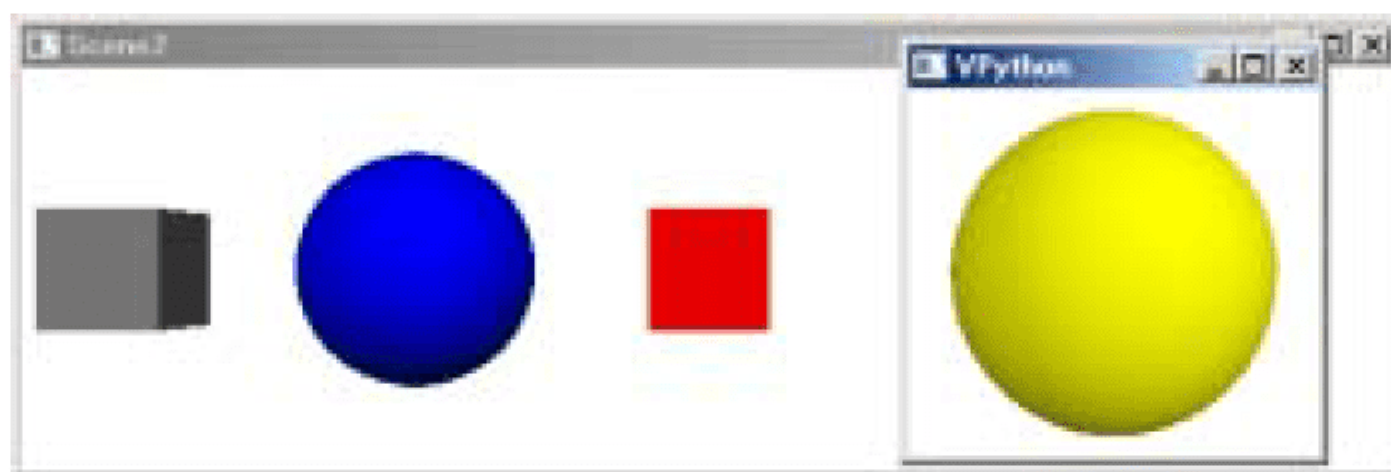


图 11-5　在第二个场景中放置球体

窗口对象有如下属性：

- foreground：在窗口中创建物体时采用的默认颜色，默认为白色。例如，运行“scene.foreground = color.green”之后，窗口中新添加物体的颜色默认为绿色。
- background：窗口的背景色，默认为黑色。
- ambient：环境光的颜色，默认为 color.gray(0.2)，gray()是计算灰度的函数，参数为 0 表示黑色，为 1 表示白色。
- lights：场景中的光源列表，场景中的默认光源为：

```
[distant_light(direction=(0.22, 0.44, 0.88), color=color.gray(0.8)),
 distant_light(direction=(-0.88, -0.22, -0.44), color=color.gray(0.3))]
```

可以用如下语句查看光源的属性：

```
>>> scene.lights[0].direction
vector(0.218217890235992, 0.436435780471985, 0.87287156094397)
```

- cursor.visible：设置场景窗口中的鼠标是否显示，False 表示隐藏鼠标。可以用它在拖曳物体时隐藏鼠标，释放物体时显示鼠标。
- objects：场景中所有可见物体的列表，被隐藏的物体和光源不在此列表之中。当设置某物体的 visible 属性为 False 以隐藏它时，其效果就是将它从此列表中删除。下面的语句使场景中所有的 box 对象都变成红色：

```
for obj in scene2.objects:
    if isinstance(obj, box):
        obj.color = color.red
```

- show_rendertime：如果值为 True，就在窗口的左下角显示诸如“cycle:27: 5”的字样。这表示场景润色的帧之间的间隔为 27 毫秒，每帧需要 5 毫秒时间润色。这还表明用户的 Python 程序每帧有 22 毫秒的处理时间。
- stereo：设置立体视觉。如果读者有双色三维立体眼镜，不妨试试这个选项。例如，设置 scene.stereo="redcyan"，就可以用红-青立体眼镜观察场景。此外还有"redblue"和"yellowblue"等选项。设置为"crosseyed"时，将产生左右两个场景，当左右眼交叉聚

焦到左右两个图时，会产生立体效果。设置为"active"时，将产生可以用主动式快门眼镜观看的立体场景。

- stereodepth：修改立体视觉的深度，默认值为 0，设置为 2 时有最好的立体效果。
- visible：窗口是否可见，当某个物体被添加进窗口时，窗口将自动被设置为可见。
- exit：为 False 时，将禁止窗口的“关闭”按钮，即无法通过“关闭”按钮关闭窗口，默认为 True。

下面所列的一些属性只能在窗口隐藏时修改。因此，通常是在用 display()创建窗口时通过关键字参数设置它们。如果需要设置可见窗口的属性，需要先通过 visible 属性将窗口隐藏。

- x、y：指定窗口在屏幕中的位置，指定的是窗口左上角的屏幕坐标。
- width、height：以像素为单位的整个窗口的宽度和高度，包括边框和标题栏。
- title：窗口标题栏中的文字，中文需要用操作系统的默认编码标准。
- fullscreen：是否采用全屏显示，如果设置为 True，将不显示窗口的边框和标题栏，按 Esc 键可退出全屏模式。

11.1.2 控制照相机

下面所列的窗口对象的属性用来控制场景中照相机的位置和方向：

- center：照相机正对的三维空间的坐标点，默认值为(0,0,0)。即使用户旋转场景(实际上是改变照相机的位置)，照相机也始终正对着这个坐标。如果修改 center 的值，照相机将保持其方向不变，进行平行移动使得其正对 center 坐标，效果如图 11-6 所示。

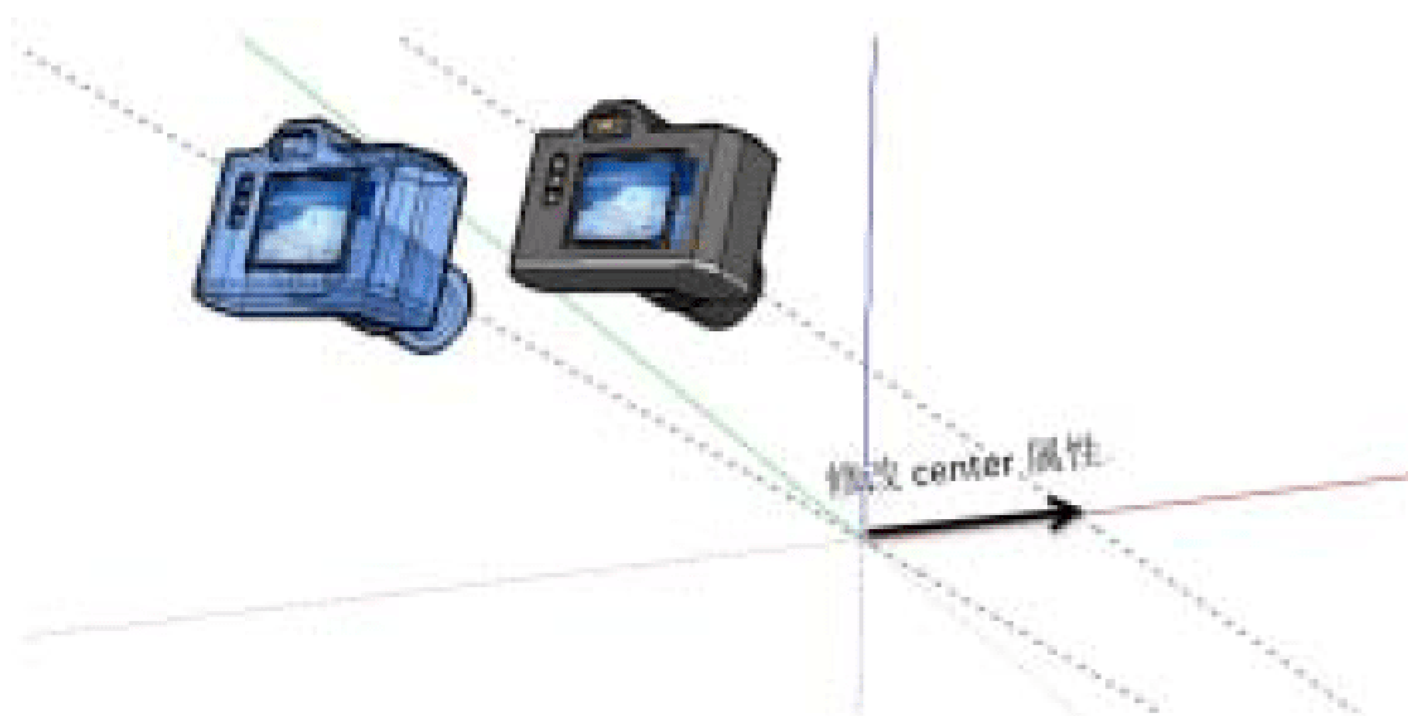


图 11-6　修改照相体的 center 属性将对照相机进行平行移动

- autocenter：如果为 True，将自动调整 center 属性，使它成为包含场景中所有物体的最小的长方体的中心，此最小长方体的各边与 X 轴、Y 轴、Z 轴平行。这样一来，照相机将始终跟随场景中的物体，因此当它为 True 时，场景中任何物体的位置发生改变，都有可能影响 center 属性。

- scale、range：它们都是三个元素的元组。range 设置照相机所能看到的范围，scale 则为 range 的倒数。例如，如果 range 为(10,10,10)，scale 将为(0.1,0.1,0.1)。
- autoscale：如果设置为 True，将根据场景中的物体自动调节，使照相机能观察到场景中的所有物体。
- forward：照相机指向的方向，默认值为(0,0,-1)，也就是从照相机所在位置到 center 坐标的方向矢量。用户不能直接修改照相机所在的位置，但可以通过 scene.mouse.camera 获得它。当用户旋转场景时，其实就是在修改 forward 属性。当 forward 被修改之后，照相机将会改变其位置，使得其方向和 forward 矢量平行。forward 的默认值为(0,0,-1)，因此是从上往下地俯视观察场景。图 11-7 演示了 forward 属性对照相机位置的影响。它将同时改变照相机的位置和方向，使照相机始终对准 center 坐标。

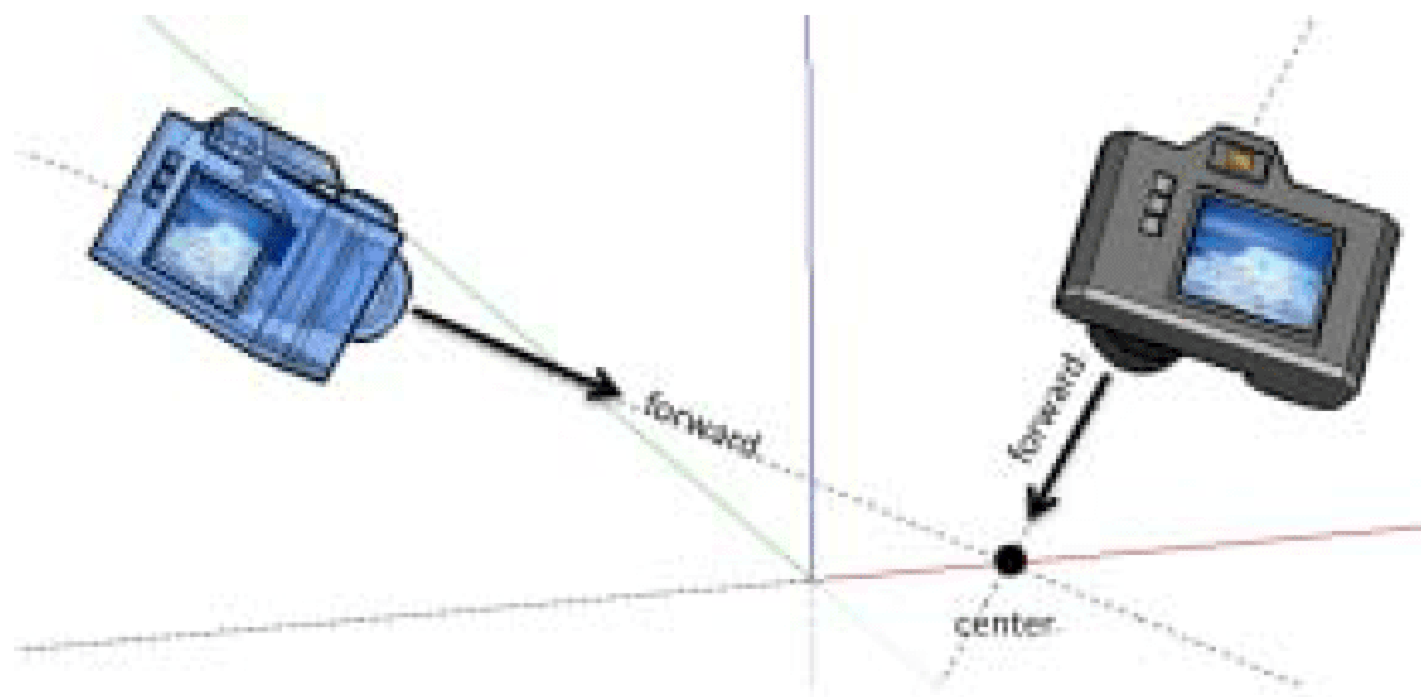


图 11-7　forward 属性可改变照相机的方向和位置

11.1.3　模型的属性

场景中的每个三维模型对象都有一些共同的属性和方法，以控制它们在场景中的位置、大小、方向及颜色等信息。下面我们以圆锥为例，详细介绍如何使用这些属性和方法。

首先调用 cone()，在场景中创建一个属性全部为默认值的圆锥：

```
>>> c = cone()
```

上面的语句会创建一个白色的、底面圆半径为 1、圆心为(0,0,0)、高为 1 的圆锥，其顶角指向 X 轴的正方向。因为场景中照相机默认沿着 Z 轴负方向观察物体，所以我们将看到一个顶角指向右方的等腰三角形。

模型的颜色由其 color 属性决定，也可以通过 red、green、blue 等属性直接访问 color 属性中的三个颜色分量。由于圆锥 c 是白色的，因此它的三个颜色分量均为 1：

```
>>> c.color
(1.0, 1.0, 1.0)
>>> c.red
1.0
```

```
>>> c.green
1.0
>>> c.blue
1.0
```

可以通过这些颜色属性修改物体的颜色，例如下面语句将圆锥改为绿色：

```
>>> c.color = (0,1,0)
```

模型的位置由 pos 属性决定，它是表示向量的 vector 对象。也可以通过模型的 x、y、z 等属性直接访问 pos 中三个轴上的分量。而 vector 对象本身也具有 x、y、z 等属性。

```
>>> c.pos
vector(0, 0, 0)
>>> c.pos = -1, 0, 0
>>> c.x
-1.0
>>> c.pos.x
-1.0
```

上面的代码将圆锥的 X 轴坐标改为-1，如果读者没有改变照相机的默认观察方向，那么会看到圆锥向左移动，它的顶点将处于场景窗口的正中心。由此可知，圆锥对象的 pos 属性决定了圆锥底面圆的中心坐标。不同模型的 pos 属性所对应的模型上的位置有所不同。例如，长方体 box 和球体 sphere 的 pos 属性指定的是它们的中心坐标。

axis 和 length 属性决定了圆锥的方向和长度。这两个属性是相互影响的，axis 属性是表示向量的 vector 对象，而 length 属性是向量的长度。不同模型的 axis 属性所表示的含义有所不同。对于圆锥来说，axis 属性是底面圆心坐标到顶点坐标的向量，即顶点坐标为“c.pos + c.axis”。

默认情况下，axis 属性为(1,0,0)，当底面圆心坐标为(-1, 0, 0)时，圆锥的顶点和坐标原点重合。length 属性就是 axis 向量的长度，向量的长度也可以通过向量的 mag 属性获得：

```
>>> c.axis
vector(1, 0, 0)
>>> c.length
1.0
>>> c.axis.mag
1.0
```

下面语句将圆锥的方向改为斜 45°，同时还改变了圆锥的长度：

```
>>> c.axis = 1, 1, 0
>>> c.length
1.4142135623730951
```

如果修改 length 属性，将保持 axis 向量的方向不变而改变其长度：

```
>>> c.length = 1
>>> c.axis
vector(0.707106781186547, 0.707106781186547, 0)
```

除了修改 axis 向量之外，还可以调用 rotate()方法改变模型的方向。rotate()将根据指定的旋转轴、旋转中心以及旋转角度对物体的 pos 和 axis 属性进行修改。下面的语句使圆锥绕过其顶点平行于 Z 轴的直线旋转 45°。angle 参数指定以弧度为单位的旋转角度，axis 参数指定旋转轴的方向，而 origin 参数指定旋转轴所经过的一点。axis 和 origin 参数的默认值分别为模型的 axis 和 pos 属性：

```
>>> c.rotate(angle=pi/4, axis=(0,0,1), origin=c.pos+c.axis)
>>> c.axis
vector(0, 1, 0)
>>> c.pos
vector(-0.292893218813453, -0.292893218813452, 0)
```

下面看一个绘制类似水雷模型的例子，效果如图 11-8 所示。模型中所有圆锥的底面圆心十分靠近球体的表面，而圆锥的方向则和坐标原点到底面圆心的方向一致。

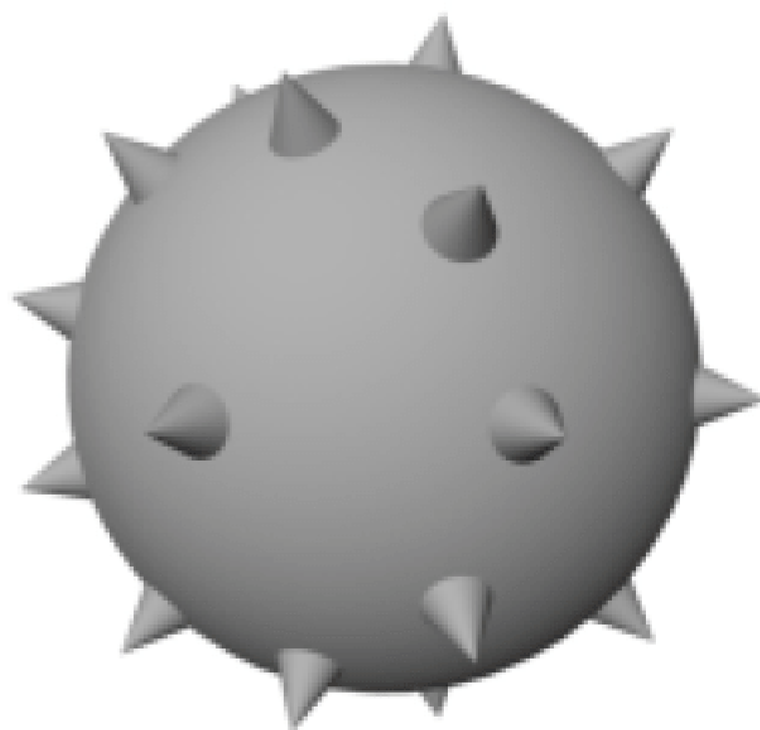
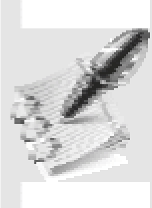


图 11-8　用球和圆锥组合而成的水雷模型



vpython_mine.py
水雷模型

```
from itertools import izip
import numpy as np
from visual import *
scene.background = 1,1,1 # 场景的背景色
color = 0.7,0.7,0.7 # 模型颜色
```

```
r = 10.0 #水雷半径

s = sphere(radius=r*1.02, color=color) ❶

# 在球坐标系中计算半径为 r 的球面上的坐标点
t, f = np.mgrid[0:pi:5j,0:2*pi:10j] ❷
xp = r*np.sin(t)*cos(f)
yp = r*np.sin(t)*sin(f)
zp = r*np.cos(t)

for pos in izip(xp.flat, yp.flat, zp.flat): ❸
    cone(pos = pos, axis=pos, length=r/5, radius=r/10, color=color) ❹
```

❶首先创建一个半径比指定半径 r 稍大的球体，模型中圆锥的底面圆心都在半径为 r 的球面上，为了让球体遮住圆锥的底面圆，需要让球体的半径大一点。❷在球坐标系中计算半径为 r 的球面上的点。❸对每个坐标点进行迭代，通过数组的 flat 属性可获得一个把数组当做一维的迭代对象，izip()则将三个迭代对象中的每个值组合成一个元组进行迭代。

❹调用 cone()创建球面上的每个圆锥。这里通过关键字参数同时指定圆锥的 pos、axis、length、radius、color 等属性。其中，radius 属性是圆锥的底面圆半径。当同时指定 axis 和 length 参数时，axis 参数只决定 axis 属性的方向，axis 属性的长度由 length 参数指定。

11.1.4 三维模型

VPython 提供了一些基本的三维模型以及创建复杂模型的方法。下面的程序在场景中展示 VPython 所提供的各种基本三维模型，效果如图 11-9 所示。



vpython_objects.py
VPython 的基本三维模型

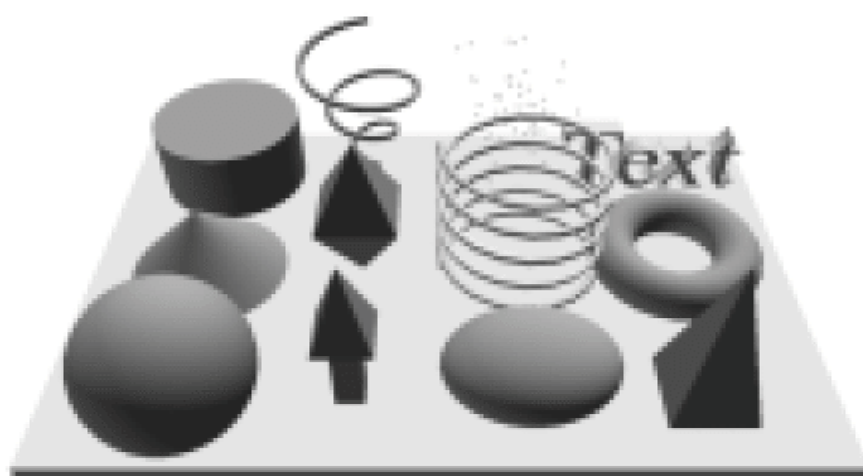


图 11-9 VPython 的基本三维模型

```
box(size=(9,6,0.2), pos=(0,0,-0.1), color=(1,1,1)) # 地板
xy = ((x,y) for x in xrange(-3,4,2) for y in xrange(-2,3,2)) ❶
idx = 0
x,y = xy.next()
sphere(pos=(x,y,0.5)) # 球体
```

```
    x,y = xy.next()
    cone(pos=(x,y,0), axis=(0,0,1)) # 圆锥
    x,y = xy.next()
    cylinder(pos=(x,y,0), axis=(0,0,1)) # 圆柱
    x,y = xy.next()
    arrow(pos=(x,y,0), axis=(0,0,2), shaftwidth=1) # 箭头
    x,y = xy.next()
    convex(pos=[(0,0,-1),(0.5,0.5,0),(-0.5,0.5,0), ❷ # 凸多面体
              (0.5,-0.5,0),(-0.5,-0.5,0),(0,0,1)], frame=frame(pos=(x,y,1)))
    x,y = xy.next()
    t = np.linspace(0,6*pi,100)
    pos =np.array([0.05*t*np.sin(t), 0.05*t*np.cos(t), t/3/pi])
    curve(pos=pos.T, radius=0.05, frame=frame(pos=(x,y,0))) ❸ # 曲线
    x,y = xy.next()
    ellipsoid(pos=(x,y,0.5), size=(2,1.5,1)) # 椭球
    x,y = xy.next()
    helix(pos=(x,y,0), axis=(0,0,2), thickness =0.1) # 弹簧、线圈
    x,y = xy.next()
    pos = np.random.normal(scale=0.5, size=(50,3))
    points(pos=pos, frame=frame(pos=(x,y,1))) ❹ # 散布点
    x,y = xy.next()
    pyramid(pos=(x,y,0), axis=(0,0,2)) # 方锥
    x,y = xy.next()
    ring(pos=(x,y,0.3), axis=(0,0,1), thickness=0.3, radius=0.8) # 圆环
    x,y = xy.next()
    text(pos=(x,y,0), text="Text", depth=0.3, up=(0,0,1), align="center") # 三维文字
```

首先，在场景中用 box()创建一个扁平的长方体作为地板。然后❶创建计算地板上每个模型位置的迭代对象 xy，每次调用“xy.next()”即可获得下一个模型的 X-Y 平面上的坐标。为了让模型的底部和地板刚好接触，不同模型的 Z 轴坐标会有所不同。

接下来是创建每个模型对象的代码。有些模型除了用 pos 属性设置其位置之外，还使用 axis 属性设置其长度和方向。此外，模型还有一些特殊的属性，例如 arrow 箭头对象有 shaftwidth 属性用来决定其粗细。关于每个模型的各种属性的含义，请读者参考 VPython 的说明手册。

下面介绍场景中几个比较特别的模型：

❷convex()创建凸多面体，它的 pos 属性是凸多面体每个顶点的坐标列表。当 pos 中的某些点使得凸多面体无法创建时，这些点将被忽略，即 convex()创建包含 pos 中所有点的凸包②。程序中创建的凸面体是一个由 6 个顶点构成的八面体。由于 pos 属性用于设置凸多面体的顶点坐标，因此缺少一个对凸多面体整体位置进行调整的参数。这里创建一个 frame 对象，并将它传递给 convex()的 frame 参数。frame 对象可以将多个模型打包在一起，让它们一起移动和旋转。这里，凸多面体的顶点坐标由其 pos 属性及其 frame 对象的 pos 属性决定。

② 关于“凸包”，有精确的数学定义，可以比较直观地理解为：凸包中任意两个点的连线都在凸包之内。

❸curve()创建三维曲线，由于曲线实际上是很细的圆管，因此可以通过 raidus 参数设置圆管的半径。曲线上的点通过曲线的参数方程计算并传递给 pos 参数。和凸多面体一样，我们用一个 frame 对象对其进行打包。❹point()创建散列点，其用法和 curve()类似。

上面的例子使用 frame 对象改变多面体、曲线和散列点的位置，每个 frame 对象中只有一个模型对象。下面我们看一个用 frame 对象修改多个模型的位置和方向的例子。



vpython_frame.py
用 frame 对象对多个模型打包

```
f = frame()
ax = arrow(axis=(1,0,0), frame=f)
ay = arrow(axis=(0,1,0), frame=f)
az = arrow(axis=(0,0,1), frame=f)

f.pos = (-0.5, -0.5, 0)
f.rotate(angle=pi/4, axis=(0,0,1))
```

程序中首先创建一个 frame 对象 f，然后创建三个坐标轴上的箭头对象，并用 frame 参数指定它们所属的 frame 对象。最后修改 frame 对象的 pos 属性，调用其 rotate()方法同时修改三个箭头对象的位置和方向。

在 IPython 中运行上面的程序之后，我们可以查看 X 轴箭头的位置和方向，发现它并没有改变：

```
>>> ax.pos
vector(0, 0, 0)
>>> ax.axis
vector(1, 0, 0)
```

为了计算 X 轴箭头的两个端点在场景中的坐标，可以调用 frame 对象的 frame_to_world()方法：

```
>>> f.frame_to_world(ax.pos) # X 轴箭头的起点坐标
vector(-0.5, -0.5, 0)
>>> f.frame_to_world(ax.pos+ax.axis) # X 轴箭头的终点坐标
vector(0.207106781186548, 0.207106781186547, 0)
```

此外，world_to_frame()方法可以将场景坐标系中的坐标转换为 frame 对象的相对坐标：

```
>>> f.world_to_frame((0,0,0))
vector(0.707106781186547, 5.55111512312578e-017, 0)
```

可以看出：场景的坐标原点在 frame 对象的 X 轴上。

11.2　制作动画演示

用 VPython 制作动画的简单之处在于：只要在一个循环体中不断地修改场景中的各个模型以及照相机的各种属性，即可实现动画效果。本节以两个实例简单地对 VPython 的动画演示功能进行介绍。

11.2.1　简单动画

首先看一个简单动画的例子，下面是完整的源程序，效果如图 11-10 所示。



vpython_simple_animation.py
球在两个板子之间来回移动的简单动画

```
from visual import *

display(title=u"简单动画".encode("gb2312"),
    width=500, height=300, background=(1,1,1))

ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red) ❶
wall_right = box(pos=(6,0,0), size=(0.1, 4, 4), color=color.green) ❷
wall_left  = box(pos=(-6,0,0), size=(0.1, 4, 4), color=color.green)

dt = 0.05 ❸
ball.velocity = vector(6, 0, 0) ❹

while True: ❺
    rate(1/dt) ❻
    ball.pos = ball.pos + ball.velocity*dt ❼
    if ball.x > wall_right.x-ball.radius or ball.x < wall_left.x+ball.radius: ❽
        ball.velocity.x *= -1
```

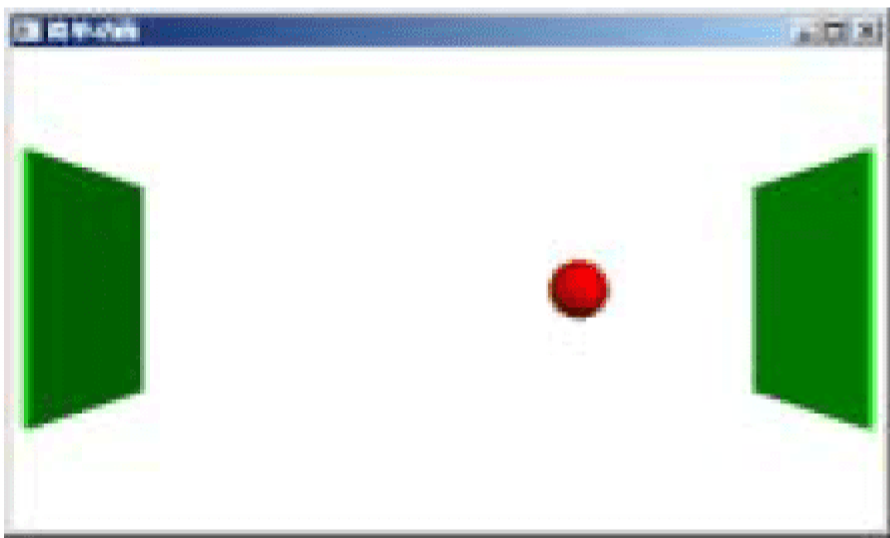


图 11-10　球在两个板子之间反复运动的简单动画

运行这段程序会出现一个有两块绿色板子和一个红球的窗口，红球在两块板子之间反复运动。

❶首先使用 sphere()创建一个红色的球体。通过 pos 和 radius 参数设置球心坐标和球体的半径。❷两块绿色板子用 box()创建，通过 size 参数设置它在 X 轴、Y 轴、Z 轴方向上的长度。使用 axis 参数也可以改变其大小，它和 size 是互相影响的。

❸用变量 dt 表示动画中两帧之间的时间间隔。❹给球体 ball 添加一个 velocity 属性，我们用它保存球体的速度。注意：velocity 不是球体对象的固有属性，而是我们为它动态添加的属性。

❺开始动画效果的循环，不断更新球体的 pos 属性，实现球体的运动效果。❻为了控制动画的播放速度，在循环中先调用 rate()，它的参数为每秒的帧数。由于 dt 为 0.05，因此动画的播放速度为每秒 20 帧。rate()会让程序等待足够长的时间，进而使动画播放的帧数接近指定的帧数。

❼修改球的 pos 属性，加上在 dt 时间段中球的位移量。❽处理球和板子的碰撞，因为 pos 为球的中心坐标，而碰撞点在球的表面，因此处理碰撞时还需要考虑球的半径。当碰撞条件满足时，只需要将球的速度矢量进行反转即可。

由于球的速度为 6，两板之间的间隔为 12，因此球从左板移到右板需要 2 秒钟时间。

11.2.2　盒子中反弹的球

下面我们看一个完整的反弹动画程序。在场景中放置 6 个半透明的盒面，形成一个立方体，球体在立方体的内部运动，程序中可以调整重力加速度(Z 轴方向的加速度)和反弹系数，同时还显示球的速度矢量和运动轨迹。运行画面如图 11-11 所示。由于程序较长，下面我们对它分段进行解释。



vpython_ball_in_box.py
球在盒子中反弹的动画

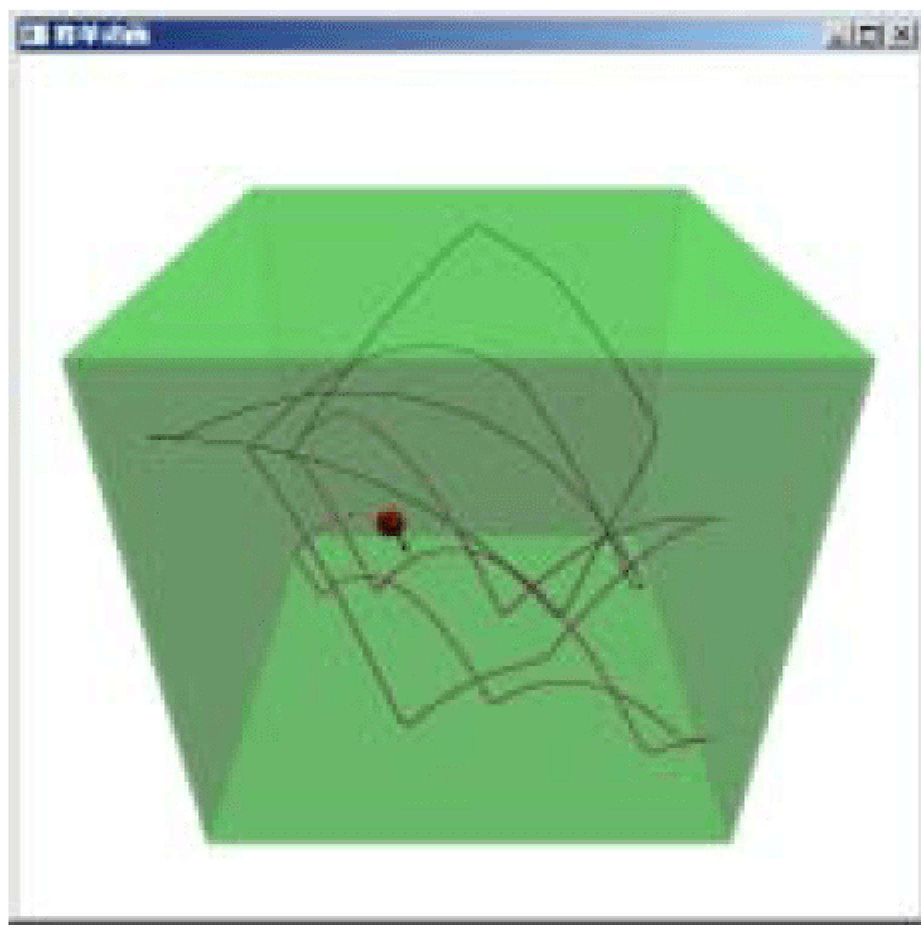


图 11-11　球在封闭盒子中反弹的动画

```
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
```

```
wall_right = box(pos=(6,0,0), size=(0.1, 12, 12), color=color.green, opacity = 0.2)
wall_left  = box(pos=(-6,0,0), size=(0.1, 12, 12), color=color.green, opacity = 0.2)
wall_front = box(pos=(0,-6,0), size=(12, 0.1, 12), color=color.green, opacity = 0.2)
wall_back  = box(pos=(0,6,0), size=(12, 0.1, 12), color=color.green, opacity = 0.2)
wall_bottom = box(pos=(0,0,-6), size=(12, 12, 0.1), color=color.green, opacity = 0.2)
wall_top   = box(pos=(0,0,6), size=(12, 12, 0.1), color=color.green, opacity = 0.2)
```

首先创建上下左右前后 6 个面以及在盒子中反弹的球体。通过 opacity 参数设置每个面的不透明度为 0.2。0.0 表示完全透明，1.0 表示完全不透明。

```
dt = 0.05
g = 9.8 # 重力加速度
f = 0.9 # 反弹能量保持系数，1.0 表示完全反弹
ball.velocity = vector(8, 6, 12) ❶
bv = arrow(pos = ball.pos, axis=ball.velocity*0.2, color=color.yellow) ❷
ball.trail = curve(color=ball.color) ❸
trail_color = 0 # 轨迹的颜色
```

定义三个常数 dt、g、f，分别表示每帧的时间间隔、重力加速度和反弹参数。❶为球体 ball 创建一个 velocity 属性，存储它的速度。❷创建一个箭头物体，它的起始点位置为球体的中心，方向和球体的速度方向相同，我们用它显示球体的速度。❸用 curve()创建一个曲线物体，并赋值给球体的 trail 属性。目前此曲线对象还是空的，没有曲线数据。我们用此曲线描绘球体的运动轨迹。初始化部分到此结束，下面的代码在 while 循环体中，对动画的每帧进行计算：

```
rate(1/dt)
# 重力加速度可改变 z 轴方向的速度，不存在反弹时修改速度
ball.velocity.z -= g * dt ❹
# 根据速度修改球体的位置
ball.pos += ball.velocity * dt ❺
```

❹使用重力加速度改变球体在 Z 轴方向的速度，然后❺根据球体的速度计算其最新的位置。

```
## 速度为正时判断正方向的盒面，速度为负时判断负方向的盒面
## 处理反弹时需要修正的球的位置，使它正好和盒面接触
# 处理左右盒面的反弹
if ball.velocity.x > 0 and ball.x >= wall_right.x - ball.radius: ❻
    ball.x = wall_right.x - ball.radius
    ball.velocity.x *= -f
if ball.velocity.x < 0 and ball.x <= wall_left.x + ball.radius:
    ball.x = wall_left.x + ball.radius
    ball.velocity.x *= -f
# 处理前后盒面的反弹
if ball.velocity.y > 0 and ball.y >= wall_back.y - ball.radius:
```

```
        ball.y = wall_back.y - ball.radius
        ball.velocity.y *= -f
    if ball.velocity.y < 0 and ball.y <= wall_front.y + ball.radius:
        ball.y = wall_front.y + ball.radius
        ball.velocity.y *= -f
    # 处理上下盒面的反弹
    if ball.velocity.z > 0 and ball.z >= wall_top.z - ball.radius:
        ball.z = wall_top.z - ball.radius
        ball.velocity.z *= -f
    elif ball.velocity.z < 0 and ball.z <= wall_bottom.z + ball.radius:
        ball.z = wall_bottom.z + ball.radius
        ball.velocity.z *= -f
```

上面的 3 个程序片段用来处理球体和盒面的碰撞，X 轴、Y 轴、Z 轴三个方向的碰撞处理方式相同，这里以 X 轴方向为例简要说明处理碰撞的算法。当球体的 X 轴方向速度为正时，判断球体是否和正方向的盒面(右侧)相撞，如果相撞就将其 X 轴方向的速度反转，并乘以碰撞系数来模拟能量损失，同时修改球体的 X 轴坐标，使其正好和右侧盒面相接触。球体的 X 轴方向速度为负时，和左侧盒面进行碰撞检测。

```
    # 更新速度箭头的位置和方向
    bv.pos = ball.pos ❻
    bv.axis = ball.velocity*0.2 ❼
    # 添加球的轨迹点
    ball.trail.append( pos = ball.pos, color = (trail_color, 0, 0)) ❽
    trail_color += 1.0/30.0*dt # 30 秒后颜色变为全红
    if trail_color > 1.0: trail_color = 1.0
```

❻❼更新箭头的位置和方向以表示球体的速度，箭头的长度和球体的速率成正比。❽将球体的当前位置添加进球体的轨迹曲线。最后更新轨迹的颜色，这样颜色将随着时间逐渐由黑变红，整个变化过程需要 30 秒时间。

11.3　与场景交互

为了和场景中的物体进行交互，VPython 提供了如下方便实用的功能：

- 键盘和鼠标事件的处理。
- 控件窗口和 4 种控件(按钮、滚动条、开关及菜单)，用于制作简单的用户界面。
- 绘图窗口，用于绘制二维坐标图。

由于篇幅受限，本书只介绍键盘和鼠标事件的处理，请读者参考 VPython 的文档和演示程序来自学其他部分的内容。

11.3.1 响应键盘事件

通过场景窗口对象的 kb 属性可以获得按键信息。kb.keys 是窗口中等待处理的键盘事件的个数，调用 kb.getkey()可以从键盘事件队列中获取一个待处理的事件。如果队列为空，getkey()将一直等待，直到产生键盘事件为止。getkey()的返回值是一个描述按键的字符串。下面是一个简单的键盘事件测试程序，可以用它查看各个按键的名称。

vpython_keyboard.py
键盘事件测试程序

```
from visual import *
keys = label()  # 用于显示文字的标签
while True:
    if scene.kb.keys: # 如果有按键按下
        s = scene.kb.getkey()  # 获得按键信息
        keys.text += s + ","
        print s
```

下面是程序输出的一些按键名称：

```
up, left, down, right, f1, f2, ctrl+a, ctrl+shift+d
```

11.3.2 响应鼠标事件

鼠标事件和键盘事件类似，通过场景窗口对象的 mouse 属性进行鼠标事件的处理。鼠标的坐标是二维视图平面上的一个点，在三维空间中有一条直线上的点都将投影到这个位置，我们称此直线为鼠标射线。scene.mouse 是一个 mouse_object 对象，下面列出它的属性和方法。为了便于理解，图 11-12 显示了鼠标射线和 pos、pickpos 等属性之间的关系。

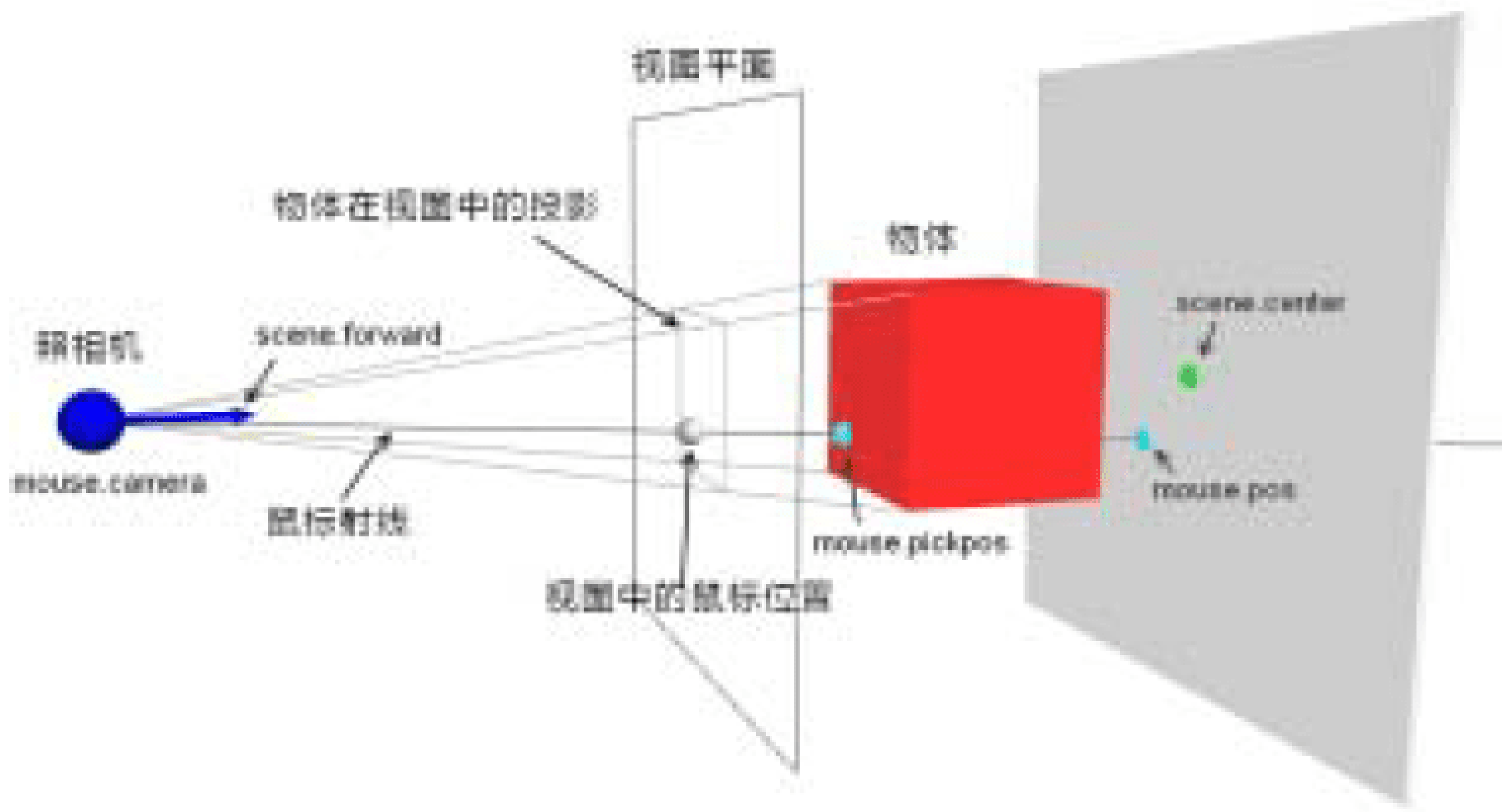


图 11-12　鼠标射线和鼠标各个坐标属性之间的关系

- pos：鼠标在三维空间中的坐标，此坐标是鼠标射线与经过点 scene.center 且平行于屏幕的平面的交点。
- button：描述鼠标按键的字符串，值可以为 None、'left'、'right'、'wheel'。此属性**只有在产生事件时才不为** None。
- pick：用鼠标选中的物体对象，与鼠标射线相交的第一个物体。
- pickpos：鼠标射线与 pick 物体的表面的交点坐标。
- camera：当前照相机的位置坐标，旋转或缩放场景时会发生变化。
- ray：从 camera 到 pos 的单位方向矢量，也就是鼠标射线的方向，它正好和窗口视图垂直。鼠标射线在三维空间中的参数方程为 camera+t*ray，其中 t 是一个大于 0 的任意参数。
- alt、ctrl、shift：Alt、Ctrl、Shift 三个按键的状态。
- project()：计算鼠标射线与任意平面的交点，平面由表示法线方向的 normal 参数和表示平面上某点坐标的 point 参数指定。因为与屏幕平行的面的法线方向为 scene.forward，所以下面语句的计算结果与 scene.mouse.pos 相同：

```
scene.mouse.project(normal=scene.forward, point=scene.center)
```

- events：待处理的鼠标事件的数目。
- getevent()：从鼠标事件队列中获取最早的鼠标事件。如果队列为空就一直等待事件的发生。

getevent()返回的事件对象保存事件发生时的鼠标坐标，也具有上述的属性和方法。除此之外，事件对象还有 press、click、drag、drop、release 等属性，它们是描述鼠标按键的字符串，分别是产生“按下”、“单击”、“拖”、“放”、“松开”5 种鼠标事件的按钮名称。


建议只使用鼠标左键作为用户交互用，因为其他按键可能会有问题。

下面是鼠标事件的演示程序：


vpython_mouse.py
鼠标事件的演示程序

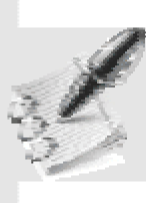
```
from visual import *
text = label(pos=(0, -2, 0))
sphere(pos=(0,2,0))
box(pos = (2, 0, 0))
ray = arrow(pos=(0,0,0), color=(1,0,0))

while True:
    rate(30)
```

```
    texts = []
    for attrname in ["pos", "pick", "pickpos", "camera", "ray"]:
        texts.append("%s=%s" % (attrname, getattr(scene.mouse, attrname)))
    texts.append("project=%s" %
      scene.mouse.project(normal=scene.forward, point=scene.center))
    text.text = "\n".join(texts)
    ray.axis = scene.mouse.ray
    if scene.mouse.events > 0:
        event = scene.mouse.getevent()
        print "press=%s, click=%s, drag=%s, drop=%s, release=%s" % (
            event.press, event.click, event.drag, event.drop, event.release
        )
```

上述程序将实时显示鼠标对象的 pos、pick、pickpos、camera、ray 等属性，并且用一个箭头表示鼠标射线的方向[3]。用 project()计算鼠标在经过点 scene.center 且平行于屏幕的平面上的投影，它的值应该和 pos 属性一致。当有鼠标事件产生时，程序输出事件对象的额外属性。

下面的程序演示了如何在场景中实现物体的拖动。



vpython_drag.py
用鼠标拖动场景中的物体

```
from visual import *

def drag_plane(mouse):
    "返回鼠标射线与 X-Y 平面的交点"
    return mouse.project(normal=(0,0,1), point=(0,0,0))

scene.range = 5 # 固定场景范围
ball = sphere(pos=(-3,0,0), color=color.cyan)
cube = box(pos=(+3,0,0), size=(2,2,2), color=color.red)
box(pos=(0,0,-1), size=(8,8,0.05))

pick = None # 当前鼠标拖动的物体

while True:
    if scene.mouse.events:
        mevent = scene.mouse.getevent() ❶
        if mevent.drag and mevent.pick: # 如果是拖动事件，并且选中了某物体
            drag_pos = drag_plane(mevent) ❷ # 拖动的起始位置
            pick = mevent.pick ❸
        elif mevent.drop: # 如果是释放事件
```

③ 由于鼠标射线的投影为一个点，因此无法直接绘制它。

```
            pick = None
    if pick:
        # 鼠标投射与 X-Y 平面的交点坐标
        new_pos = drag_plane(scene.mouse) ❹
        if new_pos != drag_pos: # 如果鼠标移动了
            pick.pos += new_pos - drag_pos ❺ # 修改物体的位置
            drag_pos = new_pos # 更新拖动的起始位置 ❻
```

❶首先用 getevent()获得要处理的鼠标事件。如果是鼠标拖动事件，并且鼠标选中了某个物体，❷就调用自定义的 drag_plane()计算出鼠标射线与 X-Y 平面的交点坐标，将其作为拖动的起始位置，❸并将当前选中的物体用 pick 变量保存。

当 pick 变量保存了某个物体时，❹调用 drag_plane()计算出当前鼠标射线与 X-Y 平面的交点坐标，❺然后根据拖动的起始位置和当前位置更新被选中物体的位置，❻并将当前位置作为下次拖动的起始位置。

这样能保证当使用鼠标选中物体的任何位置进行拖动时，物体都能够平滑地移动，不产生位置的跳变。由于 new_pos 和 drag_pos 都是鼠标射线在 X-Y 平面上的投影，因此物体的 Z 轴坐标始终保持不变，只修改 X 轴、Y 轴的坐标。通过修改 drag_plane()中调用 project()时的投影平面参数，可以在任意平面移动物体。

11.4　用界面控制场景

VPython 提供了一种控制窗口，可以放置按钮、开关及滚动条等简单控件，用以实时设置场景中的物体。但是这些控件不但功能有限，而且不是标准的界面控件，操作起来不是很方便。本节介绍如何使用 TraitsUI 制作一个能控制 VPython 场景的界面。

VPython 和 TraitsUI 各有自己的独立窗口，TraitsUI 界面有自己的消息循环，而 Visual 窗口有自己的动画控制和消息处理循环。因此我们需要使用多线程或多进程方式，让这两个循环互不影响。下面是使用多线程实现 TraitsUI 控制场景的完整程序，效果如图 11-13 所示。



图 11-13　用 TraitsUI 的界面控制 Visual 场景



vpython_traitsui.py
用 TraitsUI 的界面控制场景

```
import threading
from visual import *
```

```
from enthought.traits.api import *
from enthought.traits.ui.api import *
```

首先载入所需的模块，由于程序使用多线程实现动画和界面消息循环的分离，因此我们载入了 threading 模块。

```
class VisualTraitsUI(HasTraits):
    acceleration = Range(0.0, 20, 9.8)
    auto_scale = Bool(True)

    view = View(
        Item("acceleration", label=u"加速度"),
        Item("auto_scale", label=u"自动缩放"),
        title=u"动画控制面板"
    )
```

在 VisualTraitsUI 类中定义了两个 Trait 属性——acceleration 和 auto_scale，并在其内部创建一个 View 对象，定义界面的显示。

```
def __init__(self, *args, **kwargs):
    super(VisualTraitsUI, self).__init__(*args, **kwargs)
    self.lock = threading.Lock() ❶
    self.finish_event = threading.Event() ❷
    self.init_scene()
def init_scene(self):
    self.scene = display(title="TraitsUI Demo", background=(1,1,1))
    self.floor = box(length=4, height=0.5, width=4, color=color.blue)
    self.ball = sphere(pos=(0,4,0), color=color.red)
    self.ball.velocity = vector(0,-1,0)
    self.dt = 0.01
    self.g = self.acceleration
    self.scene.autoscale = self.auto_scale
```

在初始化方法__init__()中，❶创建了一个线程锁对象 lock，❷以及一个 Event 对象 finish_event。由于动画循环和界面事件处理不在一个线程中，为了防止两个线程同时访问同一变量时出现意想不到的问题，使用 lock 对程序进行互斥处理。当主界面关闭时，我们将使用 finish_event 通知动画循环结束运行。

在初始化场景的方法 init_scene()中创建场景。最后根据 acceleration 和 auto_scale 属性，设置场景的一些初始值：

```
def animation(self):
    while not self.finish_event.is_set(): ❸
        rate(100)
```

```
            self.lock.acquire() ❹
            self.ball.pos = self.ball.pos + self.ball.velocity*self.dt
            if self.ball.y < 1:
                self.ball.velocity.y = -self.ball.velocity.y
            else:
                self.ball.velocity.y = self.ball.velocity.y - self.g*self.dt
            self.lock.release() ❹
        self.scene.visible = False ❺
    def start_animation(self):
        self.thread = threading.Thread(None, self.animation)
        self.thread.start()
    def end_animation(self):
        self.finish_event.set() ❻
        self.thread.join() ❼
```

在处理动画循环的方法 animation()中，❸首先判断 finish_event 属性是否被设置，如果为 True 就结束动画循环。❹在循环体中，使用 lock 对象的 acquire()和 release()方法将所有处理动画计算的代码锁住。❺当动画循环结束时，设置 scene.visible 为 False，从而关闭场景窗口。

在 start_animation()中创建动画处理的线程并开始运行。在 end_animation()中，❻通过设置 finish_event 通知线程结束循环，❼然后调用 thread.join()等待线程结束。这两个方法将在界面显示之前和关闭之后调用。

```
    def _acceleration_changed(self):
        self.lock.acquire()
        self.g = self.acceleration
        self.lock.release()
    def _auto_scale_changed(self):
        self.scene.autoscale = self.auto_scale
```

当用户在界面中修改了“加速度”和“自动缩放”的设置时，acceleration 和 auto_scale 属性的值将发生改变，从而调用它们的事件处理方法。注意，我们使用 lock 对设置属性 g 的代码进行了加锁。

```
    class AnimateHandler(Handler):
        def init(self, info):
            info.object.start_animation()
            return True

        def closed(self, info, is_ok):
            info.object.end_animation()
```

界面的显示和关闭事件需要在 Handler 中进行处理，因此我们定义了一个 AnimateHandler 类。它的 init()将在界面显示之前被调用，而 closed()则在界面关闭之后调用。这两个方法的第

二个参数 info 是一个 UIInfo 对象，通过其 object 属性可以获得模型对象。

```
demo = VisualTraitsUI()
demo.configure_traits(handler = AnimateHandler())
```

最后，我们创建一个 VisualTraitsUI 对象 demo，并调用其 configure_traits()显示界面，并通过 handler 参数传递一个 AnimateHandler 对象给它。

11.5 创建复杂模型

VPython 只提供了一些简单的立体几何形状，如果要创建复杂的物体，就需要用户自己编写程序，计算物体的多边形网格模型数据，并使用 faces()将数据转换为模型进行显示。

任何一个三维模型都可以用许多三角形的面来表示，对于每个三角形的每个顶点，我们需要计算如下数据：

- 顶点的坐标：三个浮点数表示的三维坐标。
- 顶点的法线方向：三个浮点数表示的三维方向矢量。
- 顶点的颜色：三个浮点数表示的红、绿、蓝颜色分量。

将保存上述数据的三个数组传递给 faces()即可创建三维模型。对于一个有 N 个三角形的模型，每个数组的长度都是 3*3*N，也可以传递一个形状为(3*N,3)的二维数组。

11.5.1 faces()的用法

下面的例子演示了 faces()的基本用法，它通过手工设置顶点坐标、法线方向和顶点颜色等数据，绘制一个彩色三角形。



vpython_faces.py
使用 faces()绘制彩色三角形

```
pos = np.array([
    [0,0,0], # 顶点 1 的坐标
    [1,0,0], # 顶点 2 的坐标
    [0,1,0], # 顶点 3 的坐标
], dtype=np.float)

normal = np.array([
    [0,0,1], # 顶点 1 的法线方向
    [0,0,1], # 顶点 2 的法线方向
    [0,0,1], # 顶点 3 的法线方向
], dtype=np.float)
```

```
color = np.array([
    [0,0,1], # 顶点 1 的颜色
    [0,1,0], # 顶点 2 的颜色
    [1,0,0], # 顶点 3 的颜色
], dtype=np.float)

def single_face():
    faces(pos=pos, normal=normal, color=color)

single_face()
```

程序可绘制如图 11-14 所示的彩色三角形(见文前彩插)，此三角形各个顶点的数据分别是：

- 直角顶点的坐标为(0,0,0)，颜色为蓝色(0,0,1)。
- 右方顶点的坐标为(1,0,0)，颜色为绿色(0,1,0)。
- 上方顶点的坐标为(0,1,0)，颜色为红色(1,0,0)。
- 所有顶点的法线方向都是 Z 轴正方向(0,0,1)。

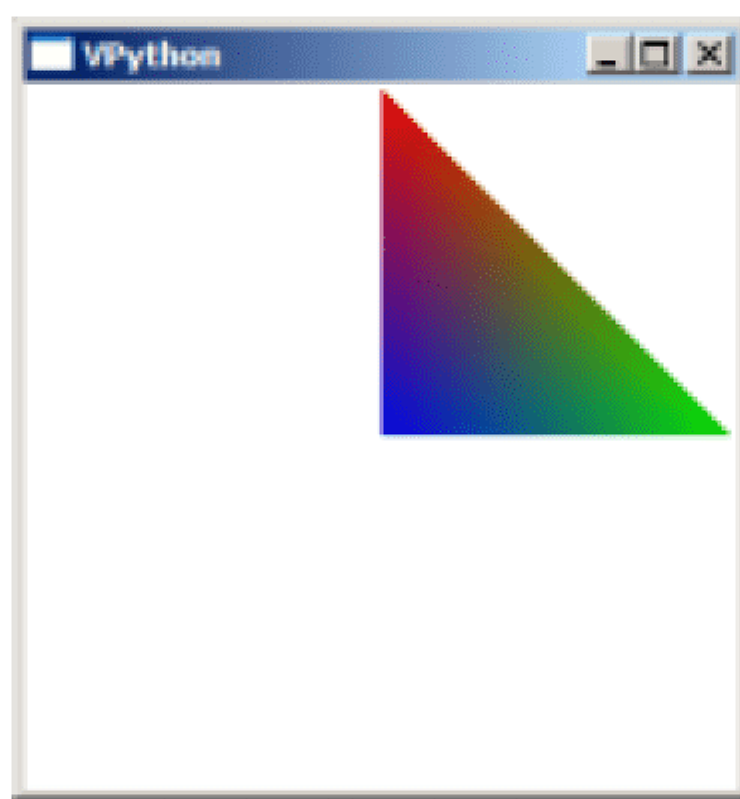


图 11-14　用 faces()绘制的彩色三角形

由于每个顶点的颜色不同，因此三角形面上的颜色为这三个顶点颜色的混合。如果旋转照相机查看此彩色三角形的反面，就会发现三角形竟然消失了。这是因为使用 faces()创建的所有三角形都是单面三角形，单面三角形只有在其顶点顺序成逆时针顺序时才会被渲染显色。上面的例子中，从 Z 轴上方观看此三角形时，其三个顶点的顺序为逆时针方向；而从 Z 轴下方观看时，顶点的顺序就变成了顺时针方向。

读者可以对调顶点 1 和顶点 2 的坐标试试，这时从 Z 轴上方就看不到三角形了。

为了实现无论从哪个方向都能看到它，我们需要创建另外一个顶点顺序相反的三角形。下面的程序创建双面三角形，这里用 pos[::-1]得到一组顺序完全相反的坐标数组，然后用 vstack()将两个二维数组沿着垂直方向(第 0 轴)叠起来，形成一个形状为(6,3)的数组，对于 normal 和 color 数组也进行同样的处理：

```
def double_face(pos, normal, color):
    triangle = frame()
    f = faces(
        pos=np.vstack([pos, pos[::-1]]),
        normal=np.vstack([normal, normal[::-1]]),
        color=np.vstack([color, color[::-1]]),
        frame=triangle)
    return triangle

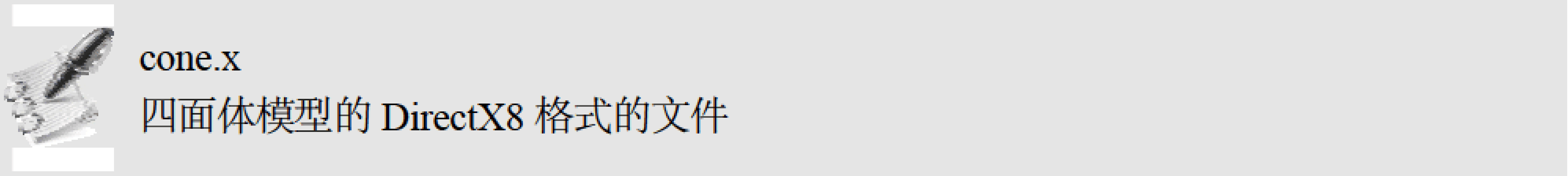
triangle = double_face(pos, normal, color)
triangle.pos = -1, -1, 0
```

在上面这段程序中，我们用一个 frame 对象对 face 对象进行封装，这样便可以通过修改 frame 对象的 pos 属性来改变三维模型的位置。

11.5.2 读入模型数据

对于复杂的模型，手工设置或计算出所有的数据是很困难的。因此，faces()通常用于显示从三维数据文件中读入的三维模型。这里以 DirectX8 格式为例，说明如何读取数据并建造复杂的模型。

首先看一下 DirectX8 的模型格式，下面是一个四面体模型的 DirectX8 格式的文件，这里只显示程序需要读入的部分：



cone.x
四面体模型的 DirectX8 格式的文件

四面体由 4 个三角形面构成，而每个三角形都有三个顶点，因此需要保存 12 个顶点坐标[④]。下面是“cone.x”文件中保存顶点坐标信息的部分。定义模型的每个面时，都通过顶点序号指定构成此面的顶点。例如第 0 个面的数据为“3; 0, 2, 1;,”，这表示它由序号为 0、2、1 的三个顶点构成，这三个顶点的坐标可在前面的顶点坐标列表中寻找：

```
Mesh {
12; // 顶点数
0.258800; -0.965900; -1.000000;,    // 顶点 0 的坐标
0.707100; 0.707100; -1.000000;,     // 顶点 1 的坐标
0.000000; 0.000000; 1.000000;,      // 顶点 2 的坐标
0.000000; 0.000000; 1.000000;,
-0.965900; 0.258800; -1.000000;,
0.258800; -0.965900; -1.000000;,
0.000000; 0.000000; 1.000000;,
0.707100; 0.707100; -1.000000;,
-0.965900; 0.258800; -1.000000;,
0.707100; 0.707100; -1.000000;,
0.258800; -0.965900; -1.000000;,
-0.965900; 0.258800; -1.000000;;
4; // 面数
3; 0, 2, 1;, // 构成第 0 面的顶点序号，3 表示此面由三个顶点构成，0、2、1 为顶点的序号
3; 3, 5, 4;,
3; 6, 8, 7;,
3; 9, 11, 10;;
```

下面是保存材质信息的部分。四面体的每个面都有不同的颜色，因此一共有 4 个材质，每

④ 四面体模型虽然只有 4 个顶点，但是模型文件为每个三角形都单独保存顶点坐标。

个面通过材质的序号指明它所使用的材质：

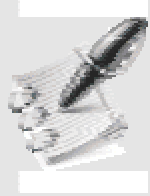
```
MeshMaterialList {
  4; // 材质数
  4; // 材质的面数
  2, // 第 0 面的材质序号
  0, // 第 1 面的材质序号
  1, // 第 2 面的材质序号
  3;;// 第 3 面的材质序号
Material Material_002 { // 材质 0
  0.882764; 0.162267; 0.067314;1.0;;  // 材质的颜色
  ...
}
Material Material {  // 材质 1
  0.150824; 0.737877; 0.810000;1.0;; // 材质的颜色
  ...
}
// 此处省略了两个材质
  }
```

最后是保存法线方向矢量的部分，它和保存坐标的方法一样。先是每个顶点的方向矢量列表，如果读者做一下计算就会发现，所有方向矢量的长度均为 1。然后是每个面的顶点序号列表，这个列表和点的坐标部分的列表是一样的：

```
////////////// 每个面的法线方向 //////////////
  MeshNormals {
12; // 顶点的法线方向矢量数，和顶点数相同
    0.228584; -0.853175; -0.468825;, // 每个顶点的法线方向矢量
    0.624561; 0.624561; -0.468825;,
    0.000000; 0.000000; 1.000000;,
    0.000000; 0.000000; 1.000000;,
    -0.853175; 0.228584; -0.468825;,
    0.228584; -0.853175; -0.468825;,
    0.000000; 0.000000; 1.000000;,
    0.624561; 0.624561; -0.468825;,
    -0.853175; 0.228584; -0.468825;,
    0.624561; 0.624561; -0.468825;,
    0.228584; -0.853175; -0.468825;,
    -0.853175; 0.228584; -0.468825;;
4; // 面数
3; 0, 2, 1;, // 面上每个顶点的法线方向的序号
3; 3, 5, 4;,
3; 6, 8, 7;,
3; 9, 11, 10;;
```

```
    } //End of MeshNormals
```

从上面的例子可以看出，文件中正好包括了我们需要读入的顶点坐标、顶点颜色和顶点法线方向三个部分。在了解三维模型的文件格式之后，我们很容易写出如下程序，进而读入文件中与模型相关的数据，并通过 faces()将其显示在 VPython 的场景中：



vpython_xmodel.py

将 DirectX8 格式的文件转为 Visual 库的模型

```
import numpy as np
from visual import *

def search_line_startswith(f, pattern):
    while True:
        line = f.readline()
        if line == "": return False
        if line.strip().startswith(pattern):
            return True

def read_directx_model(filename, double=False):
    f = open(filename)
    search_line_startswith(f, "Mesh") # 定位 Mesh 行

    point_num = int(f.readline().split(";")[0]) # 读入顶点数
    points = np.zeros((point_num, 3)) # 初始化保存顶点坐标的数组
    # 读入所有顶点坐标
    for i in xrange(point_num):
        points[i, :] = [float(x) for x in f.readline().split(";")[:3]]

    face_num = int(f.readline().split(";")[0]) # 读入面数

    # 初始化三角形的顶点坐标数组
    pfaces = zeros( (face_num*3 , 3)) # 正面的顶点下标

    # 创建所有的模型面的顶点坐标数组，将面的顶点下标替换为顶点坐标
    cnt = 0
    for i in xrange(face_num):
        p_index = [int(x) for x in f.readline().split(";")[1].split(",")]
        for j in xrange(3):
            pfaces[cnt, :] = points[p_index[j], :]
            cnt += 1

    search_line_startswith(f, "MeshMaterialList") # 定位材质
    color_num = int(f.readline().split(";")[0])  # 读入材质数
    face_color_num = int(f.readline().split(";")[0]) # 读出面数

    # 读入每个面的材质下标
```

```
face_color_list=[]
for i in xrange(face_color_num):
    face_color_list.append( int(f.readline().strip(",;\n")) )

colors = np.zeros((color_num, 3)) # 初始化存储材质颜色的数组
# 读入所有材质的颜色
for i in xrange(color_num):
    search_line_startswith(f, "Material")
    colors[i,:] = [float(x) for x in f.readline().split(";")[:3]]

# 初始化保存每个顶点颜色的数组
pcolors = np.zeros( (face_color_num*3, 3), np.float)

# 创建顶点颜色数组，将面的材质下标替换为顶点的材质颜色
cnt = 0
for i in xrange(face_color_num):
    color_idx = face_color_list[i]
    for j in xrange(3):
        pcolors[cnt, :] = colors[color_idx,:]
        cnt += 1

search_line_startswith(f, "MeshNormals") # 定位法线方向

# 读入法线方向
point_num = int(f.readline().split(";")[0])
npoints = np.zeros((point_num, 3))
for i in xrange(point_num):
    npoints[i,:] = [float(x) for x in f.readline().split(";")[:3]]

face_num = int(f.readline().split(";")[0]) # 读入面数

# 初始化顶点法线方向数组
nfaces = np.zeros( (face_num*3 , 3), np.float)

# 创建顶点法线方向数组，将每个面的法线下标替换为实际的法线方向
cnt = 0
for i in xrange(face_num):
    p_index = [int(x) for x in f.readline().split(";")[1].split(",")]
    for j in xrange(3):
        nfaces[cnt, :] = npoints[p_index[j],:]
        cnt += 1

f.close()

if double:
   pfaces = np.vstack((pfaces, pfaces[::-1]))
   nfaces = np.vstack((nfaces, nfaces[::-1]))
   pcolors = np.vstack((pcolors, pcolors[::-1]))
```

```
    model_frame = frame()
    # 使用顶点坐标数组，顶点颜色数组和顶点法线方向数组构造模型
    model = faces( pos = pfaces, normal = nfaces, color = pcolors, frame = model_frame )
    return model_frame

if __name__ == "__main__":
    import sys
    scene.background = (0.9, 0.9, 0.9)
    scene.title = sys.argv[1]
    read_directx_model(sys.argv[1], True)
```

上面程序中，为了方便操作所创建的模型，我们用一个 frame 对象将 faces 对象封装起来。如果 double 参数为 True，就使用上一节介绍的方法将所有的数据反转并添加到原始数据的后面，这样一来，所有的三角形就都是双面的，能够从任何方向看到它们。使用“vpython_xmodel.py”显示“cone.x”模型文件的效果如图 11-15 所示。

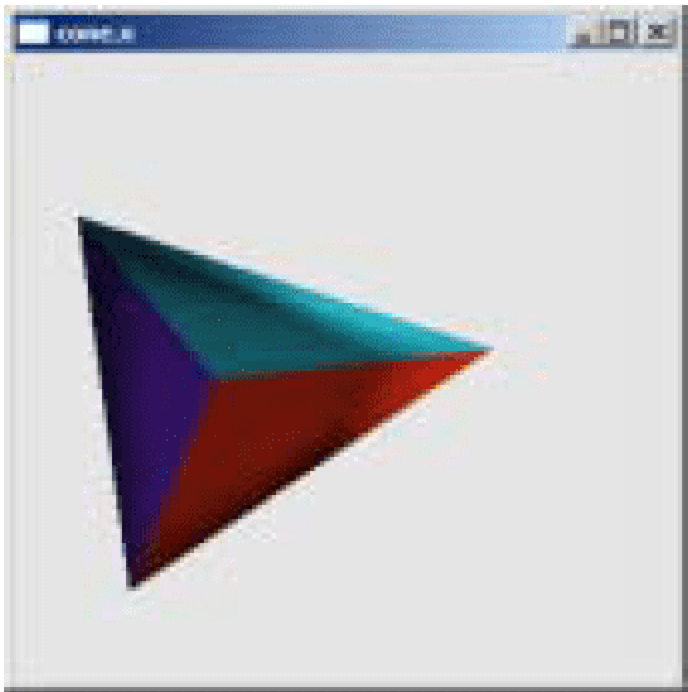


图 11-15　在 Visual 中显示 DirectX8 格式的四面体模型

我们当然不会手工创建如此复杂的模型数据文件。开源三维设计软件 Blender 支持数十种不同格式的模型文件的输入输出，其中就包括 DirectX8 格式。因此可以利用 Blender 的强大功能，转换或制作复杂的三维模型，最后用 faces 对象将其显示在动画场景中。图 11-16 显示了 Blender 设计汽车模型[⑤]时的界面及其在 VPython 场景中的显示效果。

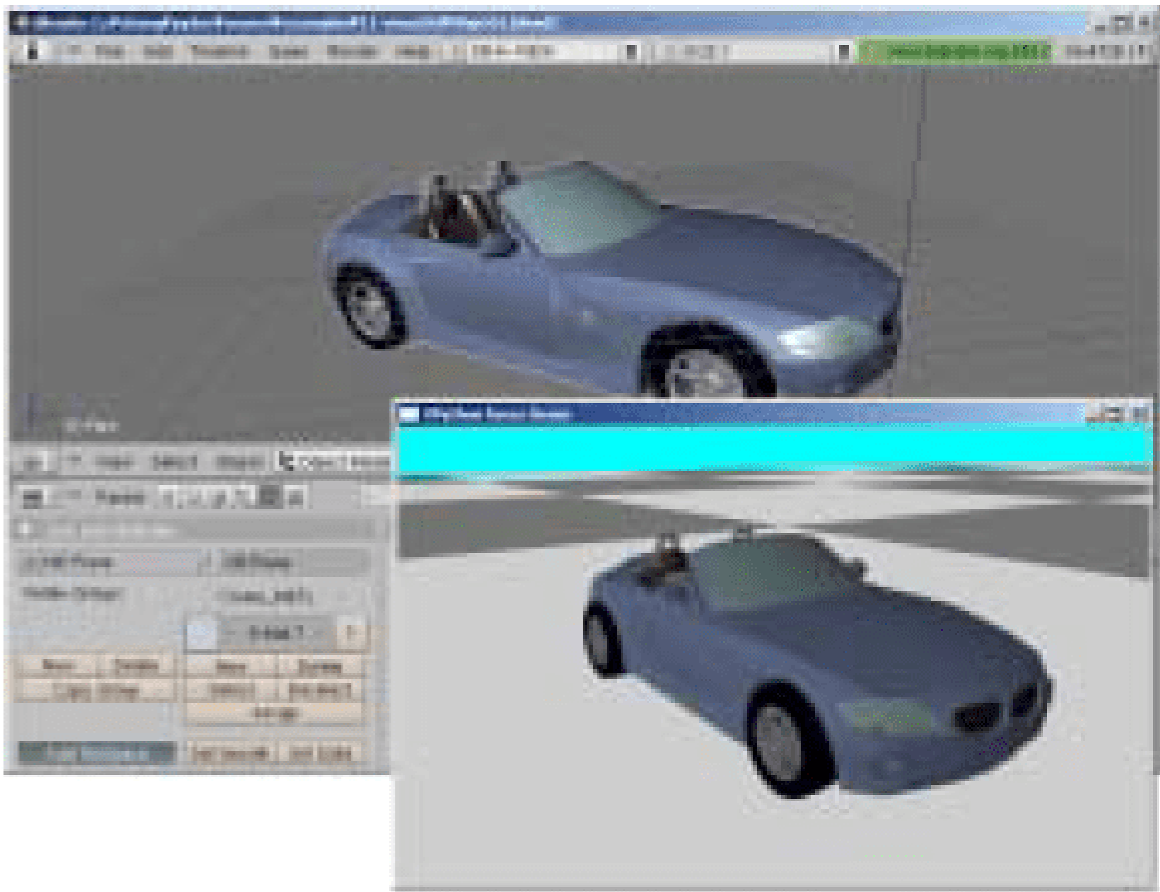


图 11-16　使用 Blender 设计汽车模型，在 VPython 中显示转换后的模型

⑤ 此汽车模型来源于互联网。

开源三维设计软件 Blender

Blender 是一个多平台的开源三维绘图和渲染软件。它提供了非常多的工具，其用户界面对初学者来说比较复杂，但花些时间阅读说明文档应该能掌握其基本用法。它采用 Python 作为脚本语言进行二次开发，因此喜欢 Python 的读者一定要下载尝试一下。Blender 的 Windows 安装版已经收录在本书所附的光盘中。



http://www.blender.org
使用 Python 作为脚本语言的开源 3D 设计软件 Blender 的官方地址

下面是汽车模型动画的演示程序。



vpython_car.py
读取 Blender 输出的汽车模型，制作汽车模拟动画

此程序除了读入并显示汽车模型之外，还制作了不断重复移动的地面和车轮的滚动效果，看上去就像汽车在地面上飞驰一样。而且可以通过按键'a'和's'控制汽车的方向[⑥]，通过按键'x'和'z'改变汽车轮胎的方向[⑦]。由于车胎需要同时围绕其 X 轴和 Z 轴旋转以达到滚动和方向转动的效果，因此车胎的旋转计算稍微有些复杂，请读者自行根据程序中的注释进行研究。

⑥ 汽车实际上是不动的，程序改变的是地面的移动方向。
⑦ 轮胎方向和汽车的转弯并不同步，实际上本程序运用于某车胎实验系统，这些参数都是直接从模拟器获得的。

第 12 章

OpenCV——图像处理和计算机视觉

OpenCV 最初是由英特尔公司开发的一套开源的跨平台计算机视觉库。它用 C 语言实现了许多图像处理和计算机视觉领域的算法，并提供了 C++的调用接口。虽然本书只介绍如何在 Python 中使用 OpenCV，但是为了更深入地了解其内部构成，推荐读者下载官方的 OpenCV 安装包，其中还包括了 PDF 文档和 OpenCV 的源程序。



http://opencv.willowgarage.com
OpenCV 的官方网址

OpenCV 的 Python 调用接口模块有如下几种：

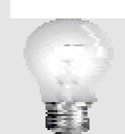
- 旧版本的 OpenCV 通过 SWIG 提供了 Python 的接口模块，最上层的模块可以通过“import opencv”载入。
- OpenCV 2.1 之后的版本则提供了一套全新的 Python 接口模块，可以通过“import cv”载入。
- 通过 ctypes 制作的接口模块载入“import ctypes_opencv”。
- 通过 Boost.Python 制作的接口模块载入“import pyopencv”。

前两套接口模块由 OpenCV 官方提供，而后两套为第三方开发。使用 Boost.Python 制作的 PyOpenCV 的 API 和 C++ API 十分相似，并且它能很好地兼容 NumPy 数组。



http://code.google.com/p/pyopencv/
PyOpenCV 项目的地址

本书只介绍 PyOpenCV 的使用方法。虽然没有详细的说明文档，但参照 OpenCV 的官方 C++文档，可以很快了解 PyOpenCV 的用法。



为了演示 PyOpenCV 中各种函数的图像处理效果，本章使用 TraitsUI 制作了许多交互式演示程序，它们的文件名均以“_demo.py”结尾。为了保证这些演示程序能够正常运行，请确认系统中正确安装了 Traits 库。



本章演示程序使用的版本为“PyOpenCV 1.2.0”，需要将 NumPy 升级到 1.4.1 版本以上才能正常使用它。

12.1 存储图像数据的 Mat 对象

为了避免命名冲突，本书使用如下方式载入 pyopencv 模块：

```
import pyopencv as cv
```

让我们从读入并显示一幅图像开始，在 IPython 中运行如下语句，将会看到一个显示美女“lena”的窗口。



opencv_show_img.py
用 OpenCV 显示一幅图像

```
>>> img = cv.imread("lena.jpg")
>>> cv.namedWindow("demo1")
>>> cv.imshow("demo1", img)
```

首先，imread()从指定的文件路径读入图像数据，它支持许多常用的图像格式，在不同的操作系统下它所支持的图像格式可能有所不同。请读者阅读 C++文档以了解 imread()的详细信息。此外，OpenCV 还提供了 imwrite()，用于将图像数据写入文件。



如果不是在交互式环境下运行上面的程序，就需在程序最后添加“cv.waitKey(0)”，否则图像窗口在显示之后将立即关闭，结束程序运行。这里，waitKey()表示等待用户按下按键，其参数为等待的毫秒数，0 表示永远等待。

为了方便用户快速观察图像处理的效果，OpenCV 提供了一些简单的 GUI 功能。这里调用 namedWindow()，创建一个名为“demo1”的窗口，最后调用 imshow()将图像显示到所创建的窗口中。imshow()的第一个参数是窗口名，第二个参数是表示图像的 Mat 对象。实际上，如果第一个参数所指定的窗口不存在，imshow()会自动创建一个新窗口，因此这里可以省略对 namedWindow()的调用。

在 OpenCV 中，Mat 对象表示图像，它是整个系统的核心，几乎所有的函数都和它相关。下面我们从它的基本属性开始逐步学习：

```
>>> type(img)
<class 'pyopencv.cxcore_hpp_point_ext.Mat'>
>>> img.size()
Size2i(width=512, height=393)
>>> img.channels()
3
>>> img.depth()
```

```
0
>>> img.type()
16
```

Mat 对象提供了 size()、channels()和 depth()等方法，可分别获得图像的大小、通道数和数值类型。上面的例子中，图像 img 的宽为 512 个像素，高为 393 个像素，有 3 个通道(channel)，即图像中每个像素的颜色用三个数值表示。每个通道的数值类型(depth)的编号为 0，0 表示无符号 8 位整数。另外还有 type()方法，可以获得一个同时表示通道数和数值类型的编号，我们可以将它理解为像素类型。关于这方面的内容将在 12.2 节详细介绍。

Mat 对象有许多属性和方法，请读者使用 IPython 的自动完成功能来查看 img 对象的属性和方法，并和 C++文档的说明进行比较。例如，我们可以找到一个 row()方法，它获得图像的指定行，返回一个新的 Mat 对象，但是它和原图像共享图像数据，这一点和 NumPy 的数组视图很相似：

```
>>> type(img.row(10))
<class 'pyopencv.cxcore_hpp_point_ext.Mat'>
>>> img.row(10).size()
Size2i(width=512, height=1)
```

12.1.1　Mat 对象和 NumPy 数组

Mat 对象本身提供的众多属性和方法并不符合 Python 的风格，因此 PyOpenCV 对 Mat 类进行了扩展，使得它能像 NumPy 的数组一样使用。下面的语句用“img[:]”从 Mat 对象 img 创建一个表示整个图像的数组，并查看数组的 shape 和 strides 属性：

```
>>>  img[:].shape
(393, 512, 3)
>>>  img[:].strides
(1536, 3, 1)
```

请注意 Mat 对象本身并不是数组，因此它没有 shape 属性：

```
>>> img.shape
AttributeError: 'Mat' object has no attribute 'shape'
```

仔细观察数组的 shape 属性，可以发现数组的第 0 轴的长度为图像的高度，第 1 轴的长度为图像的宽度，而第 2 轴的长度为图像的通道数。由 strides 属性的值可知，每个通道的数据占用 1 个字节，而一个像素点占用 3 个字节，一行数据占用 512*3=1536 个字节。因此，图像数据在内存中是连续存储的。

下面的语句获得图像的第 1 行到第 3 行、第 5 列到第 9 列、第 0 通道的数据，所得到的数组 a 和 Mat 对象共享图像数据：

```
>>> a = img[1:4,5:10,0]
>>> a.strides # 由于 a 和 img 共享图像数据，因此第 0 轴的 stride 仍然是 1536 个字节
(1536, 3)
>>> a
array([[109, 108, 107, 109, 107],
       [108, 107, 106, 110, 107],
       [106, 106, 106, 113, 108]], dtype=uint8)
```

修改 a[0,0]将同时修改 img[1,5,0]，它们在内存中的地址是相同的：

```
>>> a[0,0] = 200 # a 和 img 共享图像数据
>>> img[1,5,0]
200
```

使用下标语法也可以直接设置 Mat 对象中的数据。例如，下面将第 0 和第 1 通道的数据都设置为 0：

```
>>> img[:,:,0] = 0
>>> img[:,:,1] = 0
>>> cv.imshow("demo1", img)
```

由于在使用 imread()读入的 Mat 对象中，三个通道分别与蓝、绿、红三个颜色相对应，上面的语句将蓝、绿通道的数值都设置为 0，因此将显示一幅全是红色的图像。



实际上，在 OpenCV 内部每个通道并没有固定对应某种颜色，只是在用 imshow()、imread()和 imwrite()等函数时，才将通道按照蓝、绿、红的顺序进行输入和输出。

我们也可以使用 matplotlib 的 imshow()绘制图像，但是由于它要求图像的三个通道的存储顺序为红、绿、蓝，因此需要将图像的第 2 轴反向：

```
>>> img = cv.imread("lena.jpg")
>>> import pylab as pl
>>> pl.imshow(img[:,:,::-1]) # 将第 2 轴反向
>>> pl.show()
```

另外，使用 Mat 对象的 ndarray 属性也可以获得数组：

```
>>> img.ndarray.shape
(393, 512, 3)
```

Mat.ndarray 实际上是一个 Property 属性，每次读取它时都会调用某个函数得到一个新的数组。下面通过 id()查看 img.ndarray 的内存地址，两次结果不相同，这表示它们是不同的数组。但是请注意，这两个数组的数据存储区是相同的，都和原始的 Mat 对象 img 共享图像数据：

```
>>> img.__class__.ndarray
<property object at 0x01C47AE0>
>>> id(img.ndarray)
49696480
>>> id(img.ndarray) # 两次通过 ndarray 获得的数组不是同一个数组
49770224
```

也可以使用 asMat()函数从数组创建 Mat 对象，这样我们就能用 OpenCV 的图像处理功能对 NumPy 数组进行处理。下面的程序演示了这一过程，运行效果如图 12-1 所示(见文前彩插)。



opencv_numpy2mat.py
将数组转换为 Mat 对象并进行图像处理

```
import pyopencv as cv
import numpy as np

y, x = np.ogrid[-1:1:250j,-1:1:250j]
z = np.sin(10*np.sqrt(x*x+y*y))*0.5 + 0.5 ❶
np.round(z, decimals=1, out=z) ❷

img = cv.asMat(z) ❸

cv.namedWindow("demo1")
cv.imshow("demo1", img)

img2 = cv.Mat() ❹
cv.Laplacian(img, img2, img.depth(), ksize=3) ❺

cv.namedWindow("demo2")
cv.imshow("demo2", img2)
cv.waitKey(0)
```

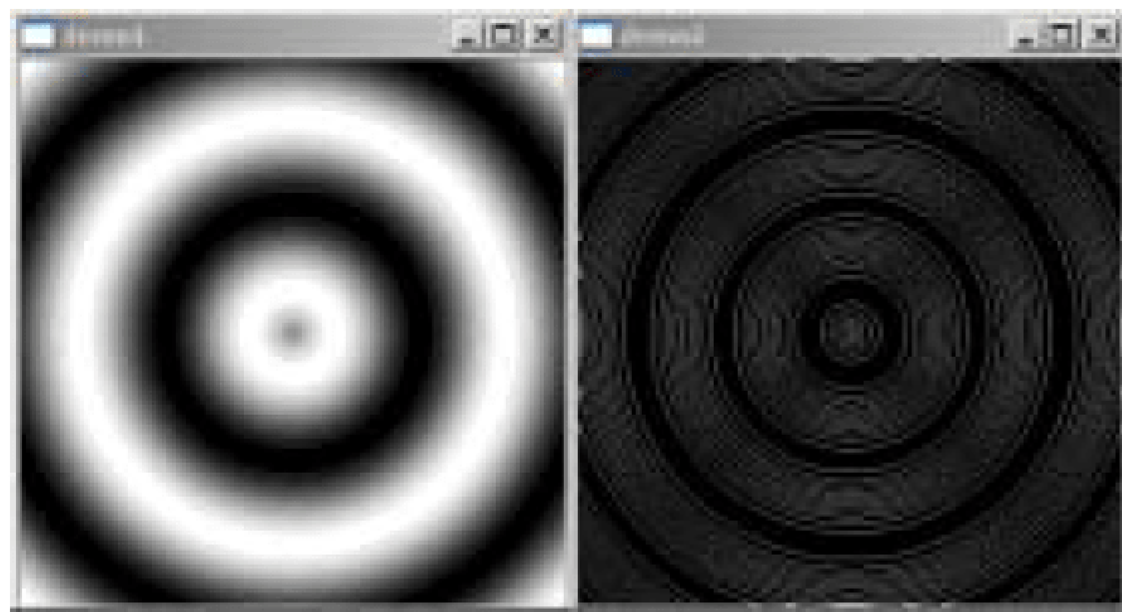


图 12-1　使用 Laplacian 算子对从 NumPy 产生的图像进行边缘检测

❶首先，数组 z 是二元函数$\frac{1}{2}\sin(10\sqrt{x^2+y^2})+\frac{1}{2}$在(-1,-1)到(1,1)网格上的计算结果，它的

取值范围是 0 到 1。❷为了使后续的图像处理函数能够正常工作，这里对数组 z 中的数值保留小数点后一位的精度。❸使用 asMat()将数组 z 转换为图像 img，图像 img 将和数组 z 共享图像数据，读者可以自行验证。

❹为了进行图像处理，先创建一个空图像 img2，当将空图像传递给某个 OpenCV 的图像处理函数时，OpenCV 会自动为其分配内存以保存结果。如果 img2 不是空图像，并且它的大小、通道数和数值类型都满足图像处理函数的输出要求，将不会分配额外的内存，而是直接把结果保存到它的图像数据区。❺调用 Laplacian()对 img 进行边缘检测，并将结果保存到 img2 中。在本例中，图像的像素类型为单通道的双精度浮点数，因此它是一幅灰度图，其中 0 表示黑色，1 表示白色。

在使用 asMat()进行转换时，它会首先尝试将数组的最后一个轴转换为图像的通道，因此用下面的语句无法将一个 4*3 的二维数组转换为单通道的 4*3 的 Mat 对象：

```
>>> a = np.zeros((4,3))
>>> b = cv.asMat(a)
```

转换后的结果是一个 1*4 的三通道 Mat 对象：

```
>>> b.size()
Size2i(width=4, height=1)
>>> b.channels()
3
```

如果设置 asMat()的 force_single_channel 参数为 True，将强制产生单通道的 Mat 对象：

```
>>> cv.asMat(a,force_single_channel=True).size()
Size2i(width=3, height=4)
```

除了 Mat 对象之外，OpenCV 还提供了 MatND 对象来表示多维数组。下面调用 asMatND()将三维数组转换成 MatND 对象，并通过 dims 属性查看其维数：

```
>>> c = cv.asMatND(np.zeros((10,20,30)))
>>> c.dims
3
```

MatND 对象使用 size 和 step 属性保存每个轴的长度和字节偏移量，它们和 NumPy 数组的 shape 和 strides 属性类似。但是这两个属性都是长度固定为 32 的整数数组。因此我们需要使用 dims 属性获取其中所需的部分：

```
>>> c.size # 得到一个长度固定为 32 的整数数组
<pyopencv.cxcore_h_ext.__array_1_int_32 object at 0x04A4C688>
```

这个数组对象可以使用正整数下标获取其中的元素，或者使用 len()获取其长度，除此之外

没有提供其他的元素存取方法。因此为了访问方便，我们可以调用 list()将其转换为列表：

```
>>> list(c.size)
[10, 20, 30, 0, 0, 0, 0, ...]
>>> list(c.size)[:c.dims]
[10, 20, 30]
>>> list(c.step)[:c.dims]
[4800, 240, 8]
```

和 Mat 对象一样，MatND 对象也可以很方便地转换为数组：

```
>>> c[:].shape
(10, 20, 30)
>>> c[:].strides
(4800, 240, 8)
>>> c[:].dtype
dtype('float64')
```

12.1.2 像素点类型

图像中的每个像素点可能有多个通道。例如，用单通道可以表示灰度图像，用红、绿、蓝 3 个通道[①]可以表示彩色图像，用 4 个通道可以表示带透明度(alpha)的彩色图像。每个通道的值可以是不同的数值类型。例如，通常的图像用 8 位无符号整数表示，而医学图像可能会用 16 位整数表示。因此，像素点的类型由通道数和通道中每个数值的类型决定。

用 Mat 对象的 type()方法可以获得图像的像素点的类型，用 channels()方法可以获得像素点的通道数，用 depth()方法可以获得表示通道数据的数值类型。type()和 depth()返回的都是一个整数序号，每个整数表示的类型都在 OpenCV 的头文件“cxtypes.h”中使用“#define”定义。下面截取其中的一部分：

```
#define CV_8U   0
#define CV_8S   1
#define CV_16U  2
#define CV_16S  3
// ...省略...
#define CV_8UC1 CV_MAKETYPE(CV_8U,1)
#define CV_8UC2 CV_MAKETYPE(CV_8U,2)
#define CV_8UC3 CV_MAKETYPE(CV_8U,3)
```

在 PyOpenCV 中，可以通过模块中的全局变量获得这些常数值。数值类型名由 3 个部分组成：

- 固定部分：“CV_”。

① 3 个通道的图像不一定是存储 RGB(红绿蓝)3 种颜色的数据，也可以是 HSV(色相、饱和度和明度)等其他表示颜色的数据。

- 数值的比特数：8、16、32、64。
- 一个描述类型的字母：“U”表示无符号整数、“S”表示符号整数、“F”表示浮点数。

像素类型则在数值类型的基础上，添加“C1”、“C2”、“C3”、“C4”等，分别表示 1 到 4 个通道的像素类型。因此，“CV_16U”表示 16 位的无符号整数，而“CV_8UC3”表示 3 个通道的 8 位无符号整数：

```
>>> cv.CV_16U
2
>>> cv.CV_8UC3
16
```

下面创建一个宽 10 像素、高 20 像素的图像，每个像素点都是 3 个通道的无符号 8 位整数，因此将其转换为 NumPy 数组将得到一个三维数组，其形状为(20,10,3)，并且数值类型为 uint8。当图像只有一个通道时，数组为二维数组。

```
>>> m = cv.Mat(cv.Size(10,20), cv.CV_8UC3)
>>> m[:].shape
(20, 10, 3)
>>> m[:].dtype
dtype('uint8')
```

12.1.3　其他数据类型

在 OpenCV 中，使用 C++的泛型模板定义各种数据类型，每种类型的模板都可以根据一些模板参数创建多种实际的数据类型。因为模板创建的类型需要经过 C++编译之后才能使用，所以在 PyOpenCV 中我们无法直接使用这些泛型模板。为了解决这个问题，PyOpenCV 对各种模板创建的实际数据类型进行了封装，下面我们具体看看这些数据类型。

Point 类型表示二维或三维空间中点的坐标，坐标值可以是整数、单精度浮点数或双精度浮点数，因此一共有 6 种 Point 类型。下面我们用 IPython 的自动补全功能显示这些 Point 类型：

```
>>> cv.Point # 按 Tab 键进行自动补全
cv.Point   cv.Point2d cv.Point2f cv.Point2i cv.Point3d cv.Point3f cv.Point3i
```

这里，cv.Point 是 cv.Point2i 的别名，后缀“2i”表示它是二维的，并且坐标值使用 32 位整数表示。此外，“3”表示三维，“f”表示单精度浮点数，“d”表示双精度浮点数。

在 IPython 下输入“cv.Point2i?”，可以看到__init__()支持的各种参数：

```
>>> cv.Point2i?
...
__init__( (object)arg1, (object)_x, (object)_y) -> None :
C++ signature :
    void __init__(_object*,int,int)
```

```
...
```

根据上面显示的参数类型，我们可以用两个整数创建 Point2i 对象：

```
>>> cv.Point2i(3, 4)
Point2i(x=3, y=4)
```

PyOpenCV 的类型检测十分严格，如果使用类型不兼容的值就会出错，例如下面用浮点数作为参数就会抛出 ArgumentError 异常：

```
>>> cv.Point2i(3.0, 4.0)
ArgumentError ...
```

甚至使用 NumPy 库中的数值类型也会出错，这一点要特别注意：

```
>>> t = np.array([3,4])
>>> type(t[0]) # 由于 t[0]是 NumPy 中的一个 int32 类型的整数
<type 'numpy.int32'>
>>> cv.Point2i(t[0], t[1]) # 因此用 t[0]作为参数会出现类型错误
ArgumentError ...
```

需要对数组元素进行类型转换，才能使用它的值来创建 Point2i 对象：

```
>>> cv.Point2i(int(t[0]), int(t[1]))
Point2i(x=3, y=4)
>>> cv.Point2i(*t.tolist())
Point2i(x=3, y=4)
```

在后面的实例程序中，我们经常需要用数组中的结果创建 PyOpenCV 中的对象，这时要特别注意类型转换问题。

和 Mat 对象一样，Point 对象可以和数组相互转换。使用 asPoint*()可以将数组转换成对应类型的 Point 对象：

```
>>> p = cv.asPoint3f(np.array([1,2,3],dtype=np.float32)) # 将数组转换成 Point 对象
>>> p
Point3f(x=1.0, y=2.0, z=3.0)
>>> p[:].dtype # 通过[:]下标获得 NumPy 数组
dtype('float32')
>>> p[:] = 10, 20, 30 # 获得的 NumPy 数组是视图
>>> p
Point3f(x=10.0, y=20.0, z=30.0)
```

Size 类型表示二维空间上的大小，一共有两种 Size 类型：Size2i 和 Size2f。用 asSize2i()和 asSize2f()可以将数组转换为相应的 Size 类型。它们的用法和 Point 类型相同，请读者自行在

IPython 下测试。

Rect 类型表示二维空间上的矩形，它只支持整数类型。

Vec 类型可以用来表示较短的一维数值数组。根据数组的长度和数值类型，它有多种版本，它们都有对应的函数从数组进行转换。

```
>>> cv.Vec # 按 Tab 键自动补全
cv.Vec2b cv.Vec2i cv.Vec3b cv.Vec3i cv.Vec4b cv.Vec4i cv.Vec6d
cv.Vec2d cv.Vec2s cv.Vec3d cv.Vec3s cv.Vec4d cv.Vec4s cv.Vec6f
cv.Vec2f cv.Vec2w cv.Vec3f cv.Vec3w cv.Vec4f cv.Vec4w
```

后缀中的数字表示所创建的数组的长度，最后的字符则表示数据类型，除了前面介绍的“i”、“f”和“d”之外，“b”表示 8 位无符号整数，“s”表示 16 位符号整数，“w”表示 16 位无符号整数。

12.1.4 Vector 类型

在 C++中，Vector 是可动态分配内存的一维数组模板，用它可以创建各种不同元素类型的动态数组类型。PyOpenCV 对 OpenCV 中用到的所有 Vector 模板创建的数组类型都进行了封装：

```
>>> cv.vector_ # 按 Tab 键自动补全
[[省略]]
cv.vector_Point2i                cv.vector_int
cv.vector_Point3d                cv.vector_int16
cv.vector_Point3f                cv.vector_int64
cv.vector_Point3i                cv.vector_int8
cv.vector_Ptr_Mat                cv.vector_long
[[省略]]
```

这些类型中的大部分都可以通过 asvector_*()函数将数组转换成对应的类型对象：

```
>>> cv.asvector_ # 按 Tab 键自动补全
[[省略]]
cv.asvector_Point2i          cv.asvector_Vec6f
cv.asvector_Point3d          cv.asvector_float32
[[省略]]
```

下面我们通过一个例子来学习 Vector 类型的用法。

用 vector_Point2i 对象可以表示二维平面上的一组点，可以用 asvector_Point2i()将一个 shape 为(N,2)的整数数组转换为 vector_Point2i 对象：

```
>>> a = np.arange(6).reshape(-1,2)
>>> p = cv.asvector_Point2i(a)
>>> p
```

```
vector_Point2i(len=3, [Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=4, y=5)])
```

Vector 对象的用法和列表类似，可以使用切片下标或者动态地改变其大小：

```
>>> p[:2]
vector_Point2i(len=2, [Point2i(x=0, y=1), Point2i(x=2, y=3)])
>>> del p[-1]
>>> p.append( cv.Point2i(10,10) )
vector_Point2i(len=3, [Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=10, y=10)])
```

tolist()方法可以将其转换成列表：

```
>>> l = p.tolist()
>>> l
[Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=10, y=10)]
```

也可以用上面的列表创建 vector_Point2i 对象：

```
>>> cv.vector_Point2i(l)
vector_Point2i(len=3, [Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=10, y=10)])
```

vector_Point2i 的初始化方法实际上会调用 vector_Point2i 类的 fromlist()来创建对象：

```
>>> cv.vector_Point2i.fromlist(l)
vector_Point2i(len=3, [Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=10, y=10)])
```

可以通过 Vector 类的 elem_type 属性获得元素的类型：

```
>>> cv.vector_Point2i.elem_type()
<class 'pyopencv.cxcore_hpp_point_ext.Point2i'>
```

最后，vector_vector_Point2i 类型是一个元素类型为 vector_Point2i 的 Vector 对象。和前面的方法类似，可以用下面的语句创建它的对象：

```
>>> cv.vector_vector_Point2i([p, p[:2]])
vector_vector_Point2i(len=2,
[vector_Point2i(len=3, [Point2i(x=0, y=1), Point2i(x=2, y=3), Point2i(x=10, y=10)]),
 vector_Point2i(len=2, [Point2i(x=0, y=1), Point2i(x=2, y=3)])])
```

12.1.5 在图像上绘图

虽然绘图功能不是 OpenCV 的重点，但是为了在图像上做必要的标识，OpenCV 提供了一些简单的绘图功能。例如，使用 line()可以在图像上绘制线段。让我们先看看它的文档帮助，在 IPython 中输入：

```
>>> cv.line?
[[省略]]
line( (Mat)img, (Point2i)pt1, (Point2i)pt2, (Scalar)color
[, (object)thickness=1 [, (object)lineType=8 [, (object)shift=0]]])
 -> None
```

img 参数的类型是 Mat，它是绘图函数的目标图像。pt1 和 pt2 参数的类型为 Point2i，分别表示线段的起点和终点。color 参数的类型为 Scalar，通过它设置线段的颜色。最后用“[]”括起来的部分是可选参数，其中 thickness 设置线段的粗细，lineType 设置线段的绘制方法：4 表示 4 连通，8 表示 8 连通，cv.CV_AA(或 16)表示反锯齿。



文档帮助中还列出了对应的 C++函数的调用参数，请读者自行对照理解。

Scalar 是一个由 4 个双精度浮点数构成的类型，line()会将直线所通过像素的每个通道值都设置为 color 参数中对应的值。Scalar 对象也可以使用 CV_RGB()来创建：

```
>>> cv.CV_RGB(255, 128, 0) # 3 个参数分别为红、绿、蓝的分量
Scalar([   0.  128.  255.    0.])
```

由上面的结果可知，Scalar 中的 4 个浮点数分别表示：蓝、绿、红和透明度。虽然颜色值有 4 个通道，但是绘图函数并不能利用透明度信息和图像上已有的像素数据进行颜色混合，它只是用 color 参数的值覆盖图像的像素值。

为了绘制半透明的图形，可以先在一个全黑的图像上进行绘图，然后将目标图像和绘图图像进行混合。下面的程序演示了这一过程，结果如图 12-2 所示(见文前彩插)。



opencv_draw.py
使用图像混合绘制半透明的直线

```
import pyopencv as cv

img = cv.imread("lena.jpg")
img2 = cv.Mat(img.size(), cv.CV_8UC4)

w, h = img.size().width, img.size().height

def blend(img, img2):
    """
    混合两幅图像，其中 img2 有 4 个通道
    """
    #使用 alpha 通道计算 img2 的混合值
    b = img2[:,:,3:] / 255.0     ❶
    a = 1 - b # img 的混合值
```

```
        #混合两幅图像
        img[:,:,:3] *= a  ❷
        img[:,:,:3] += b * img2[:,:,:3]

    img2[:] = 0
    for i in xrange(0, w, w/10):
        cv.line(img2, cv.Point(i,0), cv.Point(i, h),  ❸
            cv.Scalar(0, 0, 255, i*255/w), 5)

    blend(img, img2) ❹

    img2[:] = 0
    for i in xrange(0, h, h/10):
        cv.line(img2, cv.Point(0,i), cv.Point(w, i),
            cv.Scalar(0, 255, 0, i*255/h), 5)

    blend(img, img2)

    cv.namedWindow("Draw Demo")
    cv.imshow("Draw Demo", img)
    cv.waitKey(0)
```

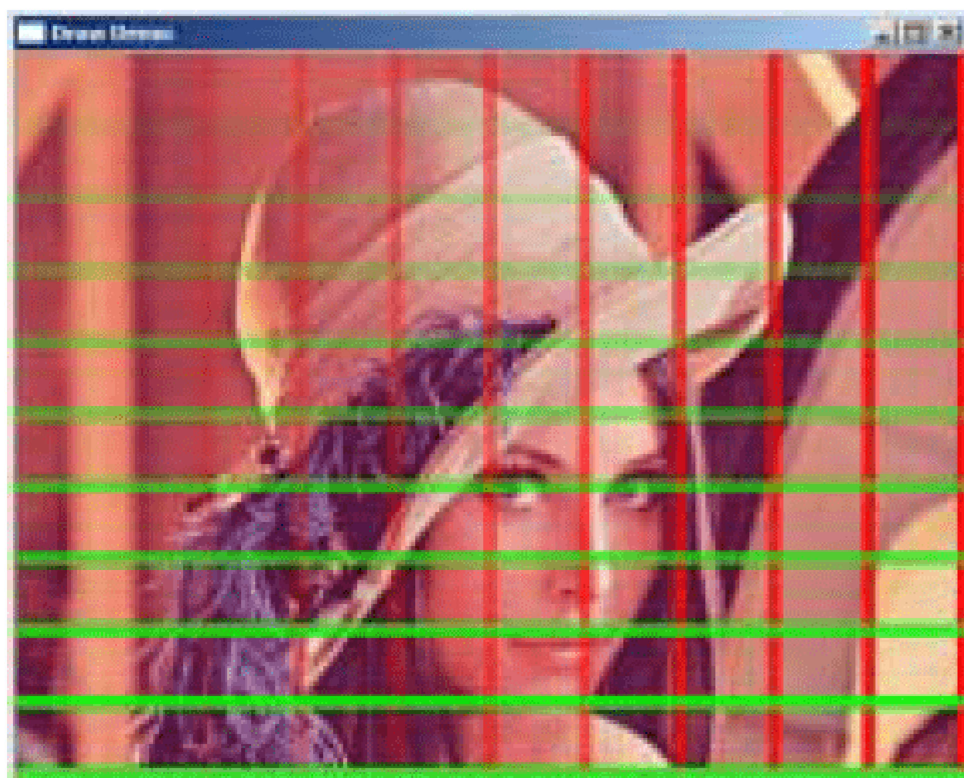


图 12-2　使用图像混合绘制半透明的直线

blend()对两幅图像进行混合，❶先用 img2 的第 4 通道计算出两个颜色混合用的数组 a 和 b。❷使用 a 和 b 将 img2 中的颜色混合到 img 中。为了减少不必要的内存分配，程序中将混合公式的计算分为两步。混合计算公式如下：

```
img 的 RGB 通道 = img 的 RGB 通道*a + img2 的 RGB 通道*b
```

❸在全黑图像 img2 上绘制竖条直线，请注意在循环中我们改变了第 4 通道的值，使得每条直线都有不同的透明度。❹最后调用 blend()，将 img2 混合到 img 中。

除了 line()之外，OpenCV 还提供了 circle()、ellipse()、rectangle()、polylines()、putText()等

绘图函数。请读者自行研究它们的用法。

12.2 图 像 处 理

OpenCV 的图像处理功能十分丰富，本节以二维卷积、形态学图像处理、颜色填充、去瑕疵为例，简要地介绍 OpenCV 的图像处理功能。希望读者通过这些实例举一反三，能通过阅读 OpenCV 的文档尝试更多的图像处理功能。

12.2.1 二维卷积

图像处理中最基本的算法就是将图像和某个卷积核进行卷积，使用不同的卷积核可以得到各种不同的图像处理效果。OpenCV 提供了 filter2D()来完成图像的卷积运算，其调用方式如下：

```
filter2D(src, dst, ddepth, kernel,
    anchor=Point2i(x=-1, y=-1), delta=0, borderType=4)
```

其中，src 参数是原始图像，dst 参数是目标图像。函数运行之后，dst 的大小和通道数将和 src 相同。ddepth 参数指定目标图像每个通道的数据类型，负数表示目标图像的数据类型和原始图像相同。kernel 参数设置卷积核，对原始图像的每个通道进行卷积计算，并将结果存储到目标图像对应的通道中。anchor 参数指定卷积核的锚点位置，当它为默认值(-1,-1)时，将以卷积核的中心为锚点。delta 参数指定在将计算结果存储到 dst 中之前，对数值的偏移量。

filter2D()的卷积运算过程如下：

(1) 对图像 src 中的每个像素点(x,y)，让它和卷积核的锚点对齐。

(2) 对于图像 src 中与卷积核重叠的部分，计算像素和卷积核值的乘积。

(3) 图像 dst 中像素点(x,y)的值为上面所有乘积的总和。

显然，当卷积核的尺寸很大时，上述方法的运算速度将会很慢。因此对于较大的卷积核，filter2D()将使用离散傅立叶变换相关的算法进行卷积运算。

下面的程序演示了使用不同卷积核对图像进行处理之后的效果，如图 12-3 所示(见文前彩插)。



opencv_filter2d.py
使用 filter2D()制作各种图像处理效果

```
import pyopencv as cv
import numpy as np
import matplotlib.pyplot as plt

# 读入图像并缩小为1/2
img0 = cv.imread("lena.jpg")
size = img0.size()
```

```
w, h = size.width, size.height
img1 = cv.Mat()
cv.resize(img0, img1, cv.Size(w//2, h//2)) ❶

# 各种卷积核
kernels = [
    (u"低通滤波器",np.array([[1,1,1],[1,2,1],[1,1,1]])*0.1),
    (u"高通滤波器",np.array([[0,-1,0],[-1,5,-1],[0,-1,0]])),
    (u"边缘检测",np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]]))
]

index = 0
for name, kernel in kernels:
    plt.subplot(131+index)
    # 将卷积核转换为 Mat 对象
    kmat = cv.asMat(kernel.astype(np.float), force_single_channel=True) ❷
    img2 = cv.Mat()
    cv.filter2D(img1, img2, -1, kmat) ❸
    # 由于 matplotlib 的颜色顺序和 OpenCV 的相反，因此需要进行反转
    plt.imshow(img2[:,:,::-1]) ❹
    plt.title(name)
    index += 1
    plt.gca().set_axis_off()
plt.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
plt.show()
```

❶为了显示方便，先调用 resize()将图像缩小为原来的二分之一。❷将使用数组定义的卷积核转换为 Mat 对象，这里使用 force_single_channel 参数保证转换之后的 Mat 对象为单通道。❸调用 filter2D()对 img1 进行卷积运算，结果写入 img2 中，并且 img2 的像素类型和 img1 的相同。❹使用 matplotlib 显示图像时，需要将 Mat 对象中通道的顺序进行反转。



图 12-3　使用 filter2D()制作的各种图像处理效果

为了对卷积核的图像处理效果有更直观的认识，读者可以运行下面的演示程序。它使用 TraitsUI 制作了一个简单的控制界面，在此界面中修改卷积核的元素值可以立即看到卷积之后的图像效果。

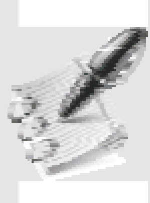


opencv_filter2d_demo.py
修改卷积核，即时观察图像处理效果

有些特殊的卷积核可以表示成一个列矢量和一个行矢量的乘积，这时只需要将原始图像按顺序与这两个矢量进行卷积即可，得到的最终结果和直接与卷积核进行卷积的结果相同。由于将一个 n*m 的矩阵分解成了两个 n*1 和 1*m 的矩阵，因此对于较大的卷积核，能大幅度地提高计算速度。OpenCV 提供了 sepFilter2D()进行这种分步卷积，其调用参数如下：

```
sepFilter2D(src, dst, ddepth, kernelX, kernelY,
  anchor=Point2i(x=-1, y=-1), delta=0, borderType=4)
```

其中，kernelX 和 kernelY 分别为行卷积核和列卷积核。下面的程序比较 filter2D()和 sepFilter2D()的计算速度：



opencv_sepFilter2D.py
用 sepFilter2D()进行高斯模糊卷积

```
import pyopencv as cv
import numpy as np
import time

img = cv.asMat(np.random.rand(1000,1000)) ❶

row = cv.getGaussianKernel(7, -1) ❷
col = cv.getGaussianKernel(5, -1)

kernel = cv.asMat(np.dot(col[:], row[:].T), force_single_channel=True) ❸

img2 = cv.Mat()
img3 = cv.Mat()

start = time.clock()
cv.filter2D(img, img2, -1, kernel) ❹
print "filter2D:", time.clock() - start

start = time.clock()
cv.sepFilter2D(img, img3, -1, row, col) ❺
print "sepFilter3D:", time.clock() - start

print "error=", np.max(np.abs(img2[:] - img3[:])) ❻
```

❶首先，随机产生一个比较大的图像 img。❷调用 getGaussianKernel()分别获得长度为 7 和长度为 5 的两个高斯模糊卷积核 row 和 col。它们的内部数据如下：

```
>>> row
Mat(rows=7, cols=1, nchannels=1, depth=6):
array([[ 0.03125 ],[ 0.109375],[ 0.21875 ],[ 0.28125 ],[ 0.21875 ],[ 0.109375],
      [ 0.03125 ]])
>>> col
Mat(rows=5, cols=1, nchannels=1, depth=6):
array([[ 0.0625],[ 0.25  ],[ 0.375 ],[ 0.25  ],[ 0.0625]])
```

❸计算 row 和 col 的矩阵乘积 kernel，它是一个形状为(5, 7)的二维数组。❹❺分别使用 filer2D()和 sepFilter2D()对图像 img 进行卷积，测量它们的计算时间，❻并输出两幅结果图像之间的最大误差。

程序的输出如下：

```
filter2D: 0.0408525080743
sepFilter3D: 0.016361184942
error= 6.66133814775e-16
```

卷积核的尺寸越大，计算时间的差别越大，请读者更改 row 和 col 的值，观察计算时间的差别。

由于卷积计算很常用，因此 OpenCV 提供了一些高级函数，以直接完成与某种特定卷积核的卷积计算。例如平均模糊 blur()、高斯模糊 GaussianBlur()、用于边缘检测的差分运算 Sobel()和 Laplacian()，等等。关于这些函数的用法请读者自行参考 OpenCV 的 C++文档，这里就不再举例了。

12.2.2 形态学运算

在第 3.6 节中，我们介绍过如何使用 SciPy 的图像处理模块进行形态学图像处理。OpenCV 也提供了类似的处理功能。例如，dilate()可以对图像进行膨胀处理，erode()则可以对图像进行腐蚀处理。另外，morphologyEx()使用膨胀和收缩实现一些更高级的形态学处理。这些函数都可以对多值图像进行操作，对于多通道图像，它们将对每个通道进行相同的运算。dilate()和 erode()的调用参数相同：

```
dilate(src, dst, kernel, anchor=Point2i(x=-1, y=-1), iterations=1, ...)
```

其中，src 参数是原始图像，dst 参数是处理之后的图像，它们的大小相同。kernel 参数是结构元素，它指定针对哪些周围像素进行计算。anchor 参数指定锚点的位置，默认值为结构元素的中心。iterations 参数指定处理次数。

morphologyEx()的参数如下：

```
morphologyEx(src, dst, op, kernel, anchor=Point2i(x=-1, y=-1), iterations=1, ...)
```

它比 dilate()多了一个 op 参数，该参数指定运算的类型。

膨胀运算可以用下面的公式描述：

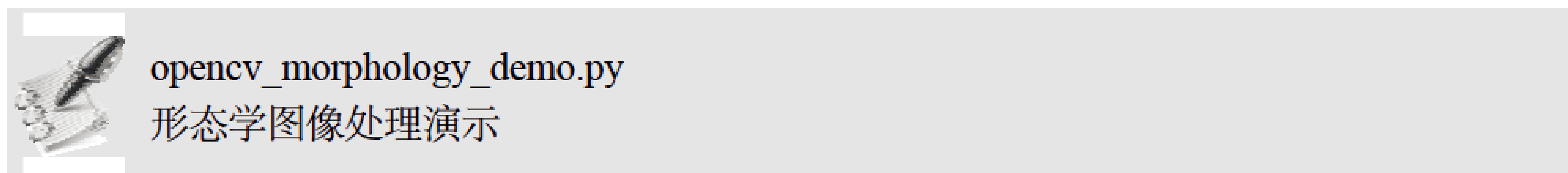
$$dst(x,y)=\max_{\text{kernel}(x',y')\neq 0} src(x+x',y+y')$$

将结构元素的锚点与原始图像中的每个像素(x,y)对齐之后，计算所有结构元素值不为 0 的像素的最大值，写入目标图像的(x,y)像素点。而腐蚀运算则是计算所有结构元素不为 0 的像素的最小值。

morphologyEx()的高级运算包括：

- MORPH_OPEN：开运算，可以用来区分两个靠得很近的区域。算法为先腐蚀后膨胀：dst=dilate(erode(src))。
- MORPH_CLOSE：闭运算，可以用来连接两个靠得很近的区域。算法为先膨胀后腐蚀：dst=erode(dilate(src))。
- MORPH_GRADIENT：形态梯度，能够找出图像区域的边缘。算法为膨胀减去腐蚀：dst=dilate(src) − erode(src)。
- MORPH_TOPHAT：顶帽运算。算法为原始图像减去开运算：dst = src-open(src)
- MORPH_BLACKHAT：黑帽运算。算法为闭运算减去原始图像：dst=close(src)-src

下面的程序演示了上述形态学图像处理的效果，图 12-4 是其界面截图。请读者通过此界面修改结构元素、处理类型以及迭代次数等参数，并观察经过处理之后的图像，理解各种运算的公式。



opencv_morphology_demo.py
形态学图像处理演示



图 12-4　形态学图像处理演示界面

12.2.3 填充——floodFill

填充函数 floodFill()在图像处理中经常用于标识或分离图像中的某些特定部分，它的调用方式有两种：

```
floodFill(image, seedPoint, newVal,
    loDiff=Scalar([ 0.  0.  0.  0.]), upDiff=Scalar([ 0.  0.  0.  0.]),
    flags=4)

floodFill(image, mask, seedPoint, newVal,
    loDiff=Scalar([ 0.  0.  0.  0.]), upDiff=Scalar([ 0.  0.  0.  0.]),
    flags=4)
```


其中，image 参数的类型为 Mat，它是需要填充的图像。seedPoint 参数的类型是 Point，它表示填充的起始点，也称为种子点。newVal 参数的类型是 Scalar，它表示填充所使用的颜色值。loDiff 和 upDiff 参数是填充的下限容差和上限容差。flags 参数是填充的算法标志。

填充从 seedPoint 参数指定的种子坐标开始，图像中与当前填充区域颜色相近的点将被添加到填充区域，从而逐步扩大填充区域，直到没有新的点能添加进填充区域为止。颜色相近的判断方法有两种：

- 默认使用相邻点为基点进行判断。
- 如果开启了 flags 中的 cv.FLOODFILL_FIXED_RANGE 标志位，那么将以种子点为基点进行判断。

假设图像中某个点(x, y)的颜色为$C(x, y)$，C_0 为基点颜色，那么当下面条件满足时，(x, y)将被添加进填充区域：

$$C_0 - loDiff \le C(x, y) \le C_0 + hiDiff$$

此外，还可以通过 flags 指定相邻点的定义：4 连通或 8 连通。

在第二种调用方式中，mask 参数是一个宽和高都比 image 大两个像素的单通道 8 位图像。image 图像中的像素(x, y)与 mask 中的(x+1, y+1)对应。填充只针对 mask 中的值为 0 的像素进行。进行填充之后，mask 中所有被填充的像素将被赋值为 1。如果只希望修改 mask，而不对原始图像进行填充，可以开启 flags 标志中的 cv.FLOODFILL_MASK_ONLY。

下面的程序演示了 floodFill()的用法，其界面截图如图 12-5 所示。在图像上用鼠标左键点选填充的种子点，通过界面上方的控件可修改 loDiff、upDiff 和 flags 等参数。



opencv_floodfill_demo.py
floodFill()填充演示程序

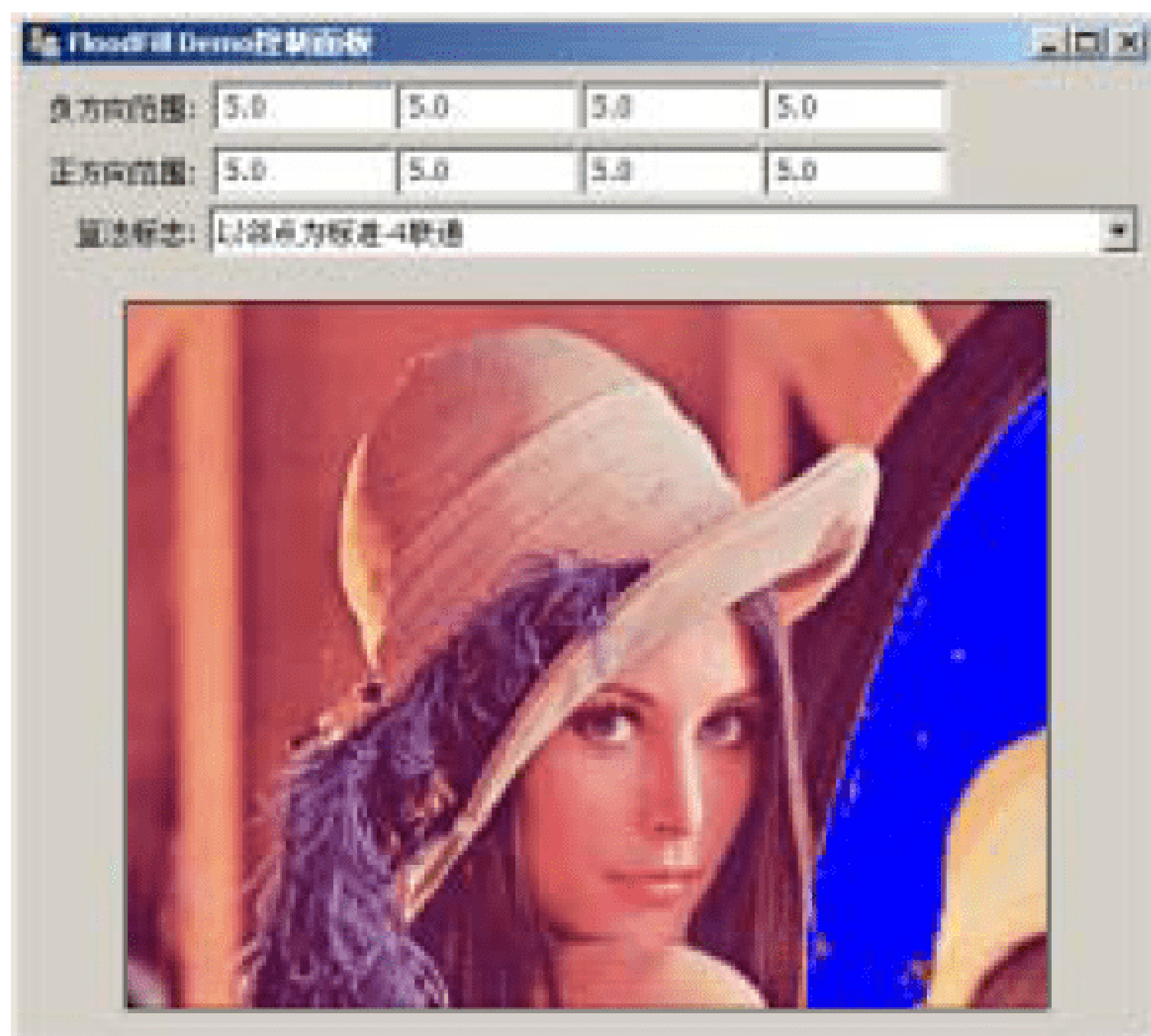


图 12-5　填充演示程序的界面截图

本程序使用 Chaco 库显示图像，并制作了一个简单的 Overlay 对象来显示填充的起始点。floodFill()中 flags 参数的选项如下：

```
Options = {
    u"以种子为标准-4 联通":cv.FLOODFILL_FIXED_RANGE | 4,
    u"以种子为标准-8 联通":cv.FLOODFILL_FIXED_RANGE | 8,
    u"以邻点为标准-4 联通":4,
    u"以邻点为标准-8 联通":8
}
```

12.2.4　去瑕疵——inpaint

使用 inpaint()可以从图像上去除指定区域中的物体，可以用于去除图像上的水印、划痕、污渍等瑕疵。它的调用参数如下：

```
inpaint(src, inpaintMask, dst, inpaintRange, flags)
```

其中，src 参数是原始图像，inpaintMask 参数是大小和 src 相同的单通道 8 位图像，其中不为 0 的像素表示需要去除的区域。dst 参数用于保存处理结果。inpaintRange 参数是处理半径，半径越大，处理时间越长，结果越平滑。flags 参数用于选择 inpaint 的算法，目前有两个候选算法——INPAINT_NS 和 INPIANT_TELEA。

下面的程序演示 inpaint()的用法，其界面截图如图 12-6(左)所示。右上图中用白色区域表示 inpaintMask 参数中不为 0 的像素，即需要处理的区域，右下图显示了对此区域进行处理之后的效果。

在此演示程序中，用鼠标在图像上绘制出需要进行处理的区域，程序会实时显示处理结果。之后可以修改“inpaint 半径”和“inpaint 算法”等设置，实时观察它们对处理结果的影响。如果选区过大，处理可能会需要较长时间，此时可以单击“保存结果”按钮，用当前的处理结果覆盖原始图像，并清除选区，以进行下一轮处理。

opencv_inpaint_demo.py
inpaint()演示程序



图 12-6　使用 inpaint()去除图像中的物体

程序中使用 Chaco 显示图像，使用自定义的 CirclePainter 对象在 Overlay 层绘制并保存鼠标轨迹，然后使用 OpenCV 的画圆函数 circle()将 CirclePainter 对象的鼠标轨迹描绘到表示处理区域的 inpaintMask 图像之上。

12.3　图 像 变 换

12.3.1　几何变换

我们可以对图像在二维平面上进行仿射变换，在三维空间中进行透视变换。仿射变换相当于将二维平面上的每个坐标点与一个 2*3 的矩阵相乘，得到新的坐标，而透视变换则是与 3*3 的矩阵相乘。原本平行的两条直线在经过仿射变换之后仍然是平行的，而经过透视变换之后，它们就可能不再平行了。

使用 warpAffine()可以对图像进行仿射变换，其调用参数如下：

```
warpAffine(src, dst, M, dsize, flags=1, borderMode=0,
           borderValue=Scalar([ 0.  0.  0.  0.]))
```

其中，src 参数是变换的原始图像，dst 参数是变换后的图像。dsize 参数为 dst 的尺寸，dst 的像素类型和 src 的相同。M 参数是仿射变换矩阵，它是一个 2*3 的 Mat 对象。flags 参数是内插方式，borderMode 是外插方式，borderValue 为背景颜色。关于这些参数的含义请读者阅读 OpenCV 的文档。

假设矩阵 M 的各个元素如下：

$$\begin{pmatrix} a_{00} \; a_{01} \; b_0 \\ a_{10} \; a_{11} \; b_1 \end{pmatrix}$$

那么仿射变换可以用下面的公式表示：

$$dst(x, y) = src(a_{00}x + a_{01}y + b_0, a_{10}x + a_{11}y + b_1)$$

为了理解仿射变换矩阵中每个参数的含义，请读者运行下面这个演示程序。



opencv_affine_demo.py
使用 TraitsUI 交互式地演示 warpAffine()的处理效果

此程序使用 TraitsUI 制作了一个可以编辑变换矩阵和图像大小的简单界面。修改这些参数时，将调用 warpAffine()实时更新图像进行仿射变换之后的效果，如图 12-7 所示(见文前彩插)。图中设置 a_{00} 和 a_{11} 为 0.5，它们使图像的宽和高都缩小为原来的二分之一。a_{01} 和 a_{10} 为 -0.3 和 0.3，当它们为符号相反、绝对值相同的两个数时，其效果相当于对图像进行旋转，否则图像将会从矩形变成平行四边形。b_0 和 b_1 则分别决定图像在水平方向和垂直方向上以像素为单位的偏移量。



图 12-7　使用 TraitsUI 交互式地演示 warpAffine()的处理效果

仿射变换矩阵中有 6 个参数，因此只需要指定变换前后 3 个坐标点的坐标，就可以通过解线性方程组获得变换矩阵。OpenCV 提供了 getAffineTransform()来帮助我们快速完成这种计算：

```
getAffineTransform(src, dst)
```

src 和 dst 参数是变换前后的三个点的坐标，它们都是 vector_Point2f 对象。下面的代码使用 getAffineTransform()计算出一个以原点为中心、放大两倍的仿射变换矩阵：

```
>>> src = cv.asvector_Point2f(np.array([[0,0],[0,1],[1,0]],dtype=np.float32))
```

```
>>> dst = cv.asvector_Point2f(np.array([[0,0],[0,2],[2,0]],dtype=np.float32))
>>> cv.getAffineTransform(src, dst)
Mat(rows=2, cols=3, nchannels=1, depth=6):
array([[ 2.,  0.,  0.],
       [ 0.,  2.,  0.]])
```

warpPerspective()和 warpAffine()类似，也对图像进行几何变换，不过它是在三维空间中进行透视变换，因此它的变换矩阵是 3*3 的矩阵。这个变换矩阵可以通过 getPerspectiveTransform()计算。

下面是 warpPerspective()的演示程序，其运行效果如图 12-8 所示(见文前彩插)。图中的变换矩阵使用 getPerspectiveTransform()计算，它的变换源坐标点分别为图像的左上、右上、左下和右下 4 个点，而变换目标点为界面上所输入的坐标。所得到的变换矩阵将原始图像 4 个角上的点变换为图中箭头所指向的 4 个点。由于程序的结构和 warpAffine()的仿射变换演示程序类似，这里就不再详细解释了。

opencv_warpPerspective_demo.py
使用 warpPerspective()对图像进行三维透视变换



图 12-8　使用 warpPerspective()对图像进行三维透视变换

12.3.2　重映射——remap

图像的各种变换都有一个共同特点：它们从原始图像上的某个位置取出一个像素点，并把它绘制到目标图像上的另外一个位置。从原始坐标到目标坐标的映射不一定是一对一的关系。OpenCV 提供了一个通用的图像映射函数 remap()来完成这种计算。remap()的参数如下：

```
remap(src, dst, map1, map2, interpolation, borderMode=0,
    borderValue=Scalar([ 0.  0.  0.  0.]))
```

其中，src 参数是原始图像，dst 参数是目标图像，最终生成的目标图像的大小和像素类型都和 src 相同。map1 和 map2 参数是两个大小与原始图像相同的 Mat 对象，它们的元素值是图像 dst 中对应下标的像素点在图像 src 中的坐标值，其元素类型可以是整数或单精度浮点数。map1 中存储映射的 X 轴坐标，map2 中存储映射的 Y 轴坐标。下面的数学公式表示了这种映射关系，其中，x 和 y 是目标图像中每个像素的坐标，通过 map1 和 map2 分别获得它们在 src 中的坐标：

$$dst(x, y) = src(map1(x, y), map2(x, y))$$

下面的程序使用 remap()将原始图像缩小为原来的二分之一：



opencv_remap_resize.py
使用 remap()缩小图像

```
img = cv.imread("lena.jpg")
size = img.size()
w, h = size
img2 = cv.Mat()
map1, map2 = np.meshgrid(
    np.linspace(0,w*2,w).astype(np.float32),
    np.linspace(0,h*2,h).astype(np.float32),
)
map1 = cv.asMat(map1)
map2 = cv.asMat(map2)
cv.remap(img, img2, map1, map2, cv.INTER_LINEAR)
```

缩小之后的图像 img2 的大小仍然和原始图像 img 相同，但是其中只有左上部分有图像数据，因此缩小为原始图像的二分之一。为了直接创建两个元素为单精度浮点数的映射数组 map1 和 map2，这里使用和 np.mgrid 对象功能类似的 np.meshgrid()函数。请读者查看 meshgrid()的函数说明以了解其使用方法。

为了演示 remap()的强大功能，下面的程序让用户输入一个三维空间的曲面函数，程序将根据此函数计算的曲面对图像进行变形，效果如图 12-9 所示，就像将图像贴在曲面上一样(见文前彩插)。程序较长，这里只给出计算 map1 和 map2 数据的代码：



opencv_remap_demo.py
使用三维曲面和 remap()对图像进行变形

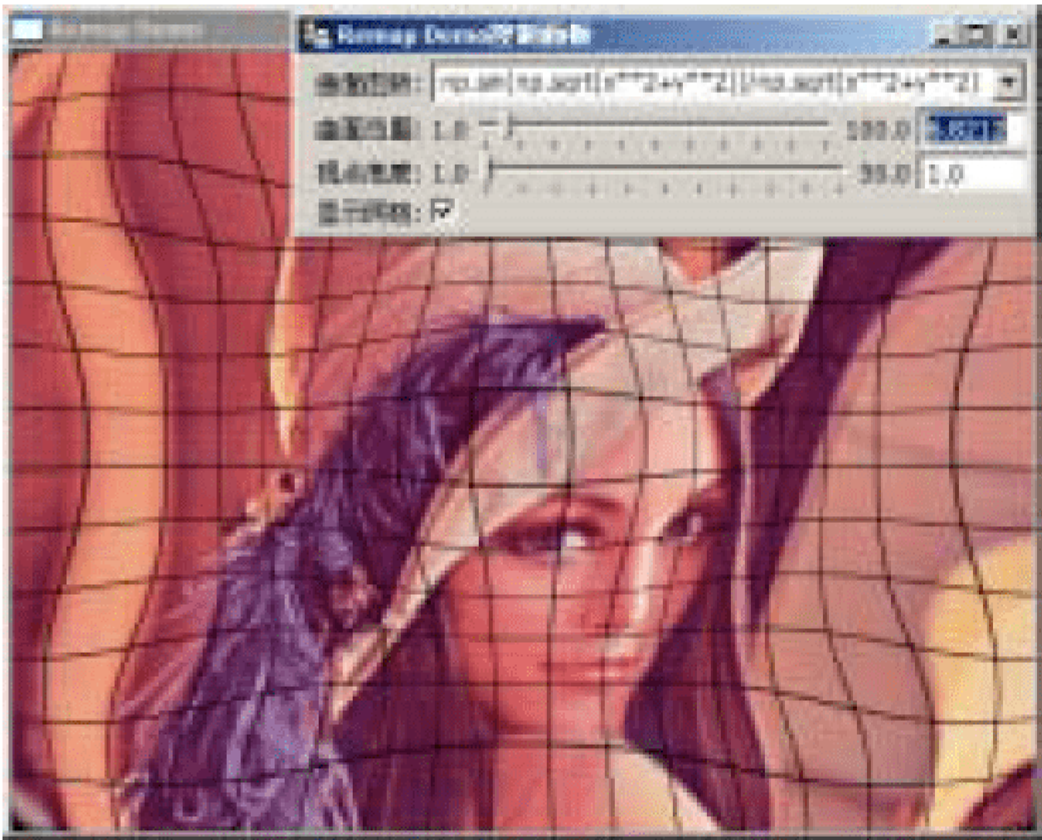


图 12-9　使用三维曲面和 remap()对图像进行变形

```
def make_surf_map(func, r, w, h, d0):
    """计算曲面函数 func 在[-r:r]范围内的值，并进行透视投影
    视点高度为曲面高度的 d0 倍+1"""
    y, x = np.ogrid[-r:r:h*1j, -r:r:w*1j]
    z = func(x,y) ❶
    d = d0*np.ptp(z) + 1.0 ❷
    map1 = x*(d-z)/d ❸
    map2 = y*(d-z)/d
    return (map1 / (2*r) + 0.5) * w, (map2 / (2*r) + 0.5) * h ❹
```

上述代码片段中，先计算指定范围内的网格 x、y，❶然后计算出网格上每一点的曲面高度 z。❷通过曲面的高度范围和 d0 参数决定观察点的高度。❸然后利用投影变换公式，计算出表示坐标变换的两个数组 map1 和 map2。其中，投影公式的示意图如图 12-10 所示。❹最后将 map1 和 map2 的取值范围改为图像的范围。

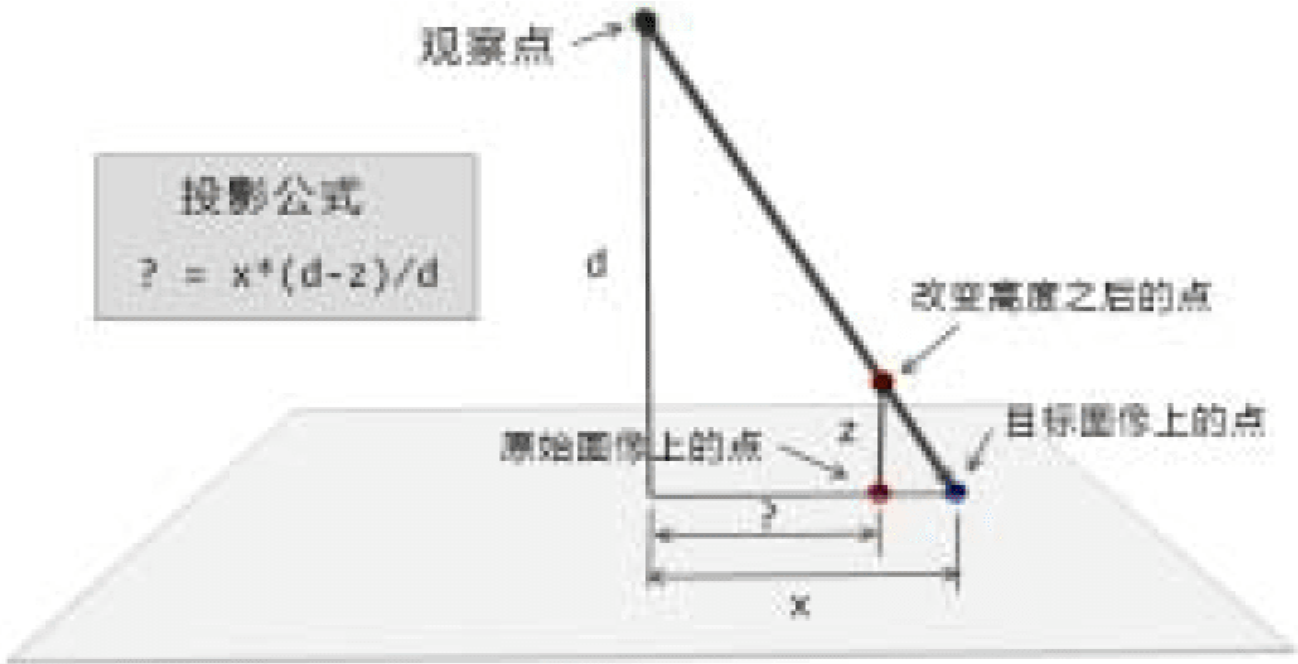


图 12-10　投影公式示意图

12.3.3　直方图统计

在 NumPy 中，有 3 个直方图统计函数——histogram()、histogram2d()和 histogramdd()，分别对应一维数据、二维数据及多维数据的情况。下面的程序用 histogram()和 histogram2d()对图像的颜色分布进行统计，并输出如图 12-11 所示的统计结果(见文前彩插)。



opencv_hist_numpy.py
用 NumPy 的直方图统计函数进行一维和二维统计

```
import pyopencv as cv
import numpy as np
import matplotlib.pyplot as plt

img = cv.imread("lena.jpg")

plt.subplot(121)
for i in xrange(3):
    hist, x = np.histogram(img[:,:,i].flatten(), bins=256, range=(0,256)) ❶
    plt.plot(0.5*(x[:-1]+x[1:]), hist, label="Ch %d" % i, lw=i+1)
plt.legend(loc="upper left")
plt.xlim((0,256))

hist2, x2, y2 = np.histogram2d(  ❷
    img[:,:,0].flatten(), img[:,:,2].flatten(),
    bins=(100,100), range=[(0,256),(0,256)])

plt.subplot(122)
plt.imshow(hist2, extent=(0,256,0,256), origin="lower")
plt.ylabel("Ch0")
plt.xlabel("Ch2")
plt.show()
```

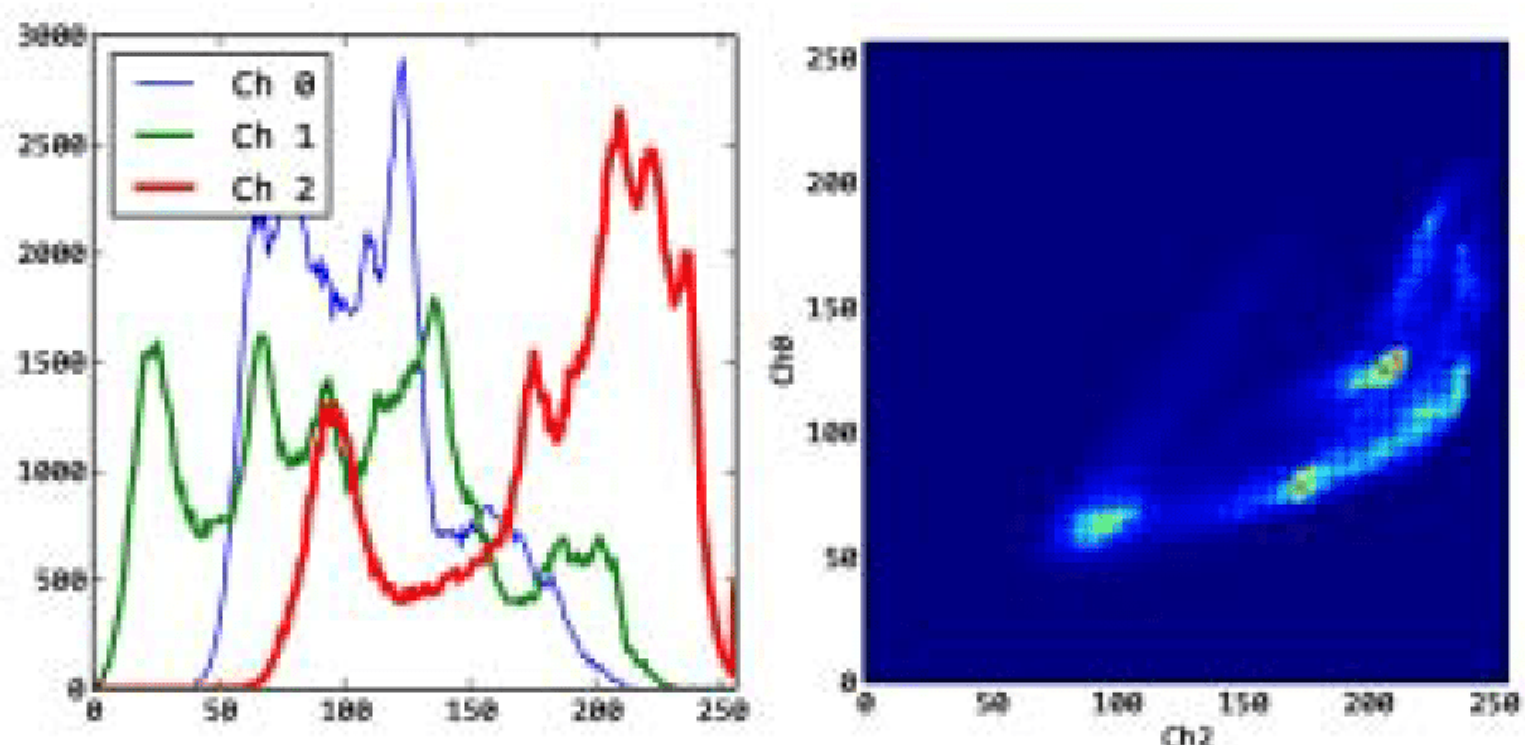


图 12-11　“lena.jpg”的 3 个通道的直方图统计(左)、通道 0 和通道 2 的二维直方图统计(右)

❶通过 histogram()对图像 img 的 3 个通道分别进行一维直方图统计，由于被统计的数组必须是一维的，因此这里调用数组的 flatten()方法将二维数组转换为一维数组。通过 range 参数指定统计区间为 0 到 256，通过 bin 参数指定将统计区间等分为 256 份。histogram()返回两个数组 hist 和 x，其中 hist 为统计结果，长度为 bin；x 为统计区间，长度为 bin+1。hist[i]的值为数组中“>=x[i]”且“<x[i+1]”的元素的个数。

❷用 histogram2()对通道 0 和通道 2 进行二维直方图统计。被统计的数组是两个一维数组，因此也需要用 flatten()方法进行转换。它们分别为图像中通道 0 和通道 2 的数据。bins 和 range 参数都变成了包含两个元素的序列，分别与两个数组相对应。返回的统计结果 hist2 是一个二维数组，其形状由 bins 决定。第 0 轴与第一个数组相对应，第 1 轴与第二个数组相对应。它是对由两个一维数组的对应元素所构成的二维矢量的分布统计结果。

观察图 12-11(左)可知，通道 2(红色通道)的值较大，因此整个图像呈现暖色调。而从右图不但可以得出通道 2 的值比通道 0 较大的结论，还可以看到几处分布比较密集的领域。例如，其中一块的中心坐标大约为(207, 125)，这说明图像中红色值在 207 附近、蓝色值在 125 附近的像素点较多。

OpenCV 中的直方图统计函数为 calcHist()。它支持对多幅图像进行 N 维直方图统计，因此其调用参数比较复杂。下面是使用 calcHist()对一副图像进行二维直方图统计的例子：



opencv_hist_cv.py
使用 calcHist()进行二维直方图统计

```
import pyopencv as cv
import numpy as np

img = cv.imread("lena.jpg")
result = cv.MatND()

r = cv.vector_float32([0, 256]) ❶
ranges = cv.vector_vector_float32([r, r])

cv.calcHist(cv.vector_Mat([img]), ❷
    channels = cv.vector_int([0, 1]),
    mask = cv.Mat(),
    hist = result,
    histSize=cv.vector_int([30, 20]),
    ranges=ranges
    )

hist, _x, _y = np.histogram2d(img[:,:,0].flatten(), img[:,:,1].flatten(),
    bins=(30,20), range=[(0,256),(0,256)])

print np.all(hist == result[:]) ❸
```

❷calcHist()的第一个参数是一个元素类型为 Mat 图像的 Vector 对象，它将对其中的所有图像进行统计。channels 参数是一个整数 Vector 对象，指定要统计的通道，其长度决定了直方图的维数。mask 参数是一个 Mat 对象，可以用它对图像进行遮罩，从而只统计图像中的部分区域，本例中使用空 Mat 对象，表示对图像上所有的点进行统计。hist 参数指定直方图的统计结果，因为它可以存储任意维数的统计结果，因此它是一个 MatND 对象。histSize 参数是一个整数 Vector 对象，用来指定各个通道的等分区域数，和 histogram2d()的 bins 参数意义相同。

❶ranges 参数和 histogram2d()的 range 参数意义相同，不过它是一个类型为 vector_vector_float32 的 Vector 对象，创建起来比较费事。它的每个元素都是一个包含两个单精度浮点数的 Vector 对象。❸最后和 histogram2d()的计算结果进行比较。

计算出直方图之后，我们可以用 calcBackProject()将图像中的每点替换为它在直方图中对应的值。于是，在直方图中出现次数越高，图像中对应的像素就越亮。我们可以用这种方法找出图像中和直方图相匹配的区域。让我们用一个实际的例子来说明：



opencv_back_project.py
使用 calcBackProject()寻找颜色近似的区域

程序的输出如图 12-12 所示。它在图像“fruits.jpg”(上中图)中寻找和图像“fruits_section.jpg”(上左图)颜色近似的部分(见文前彩插)。

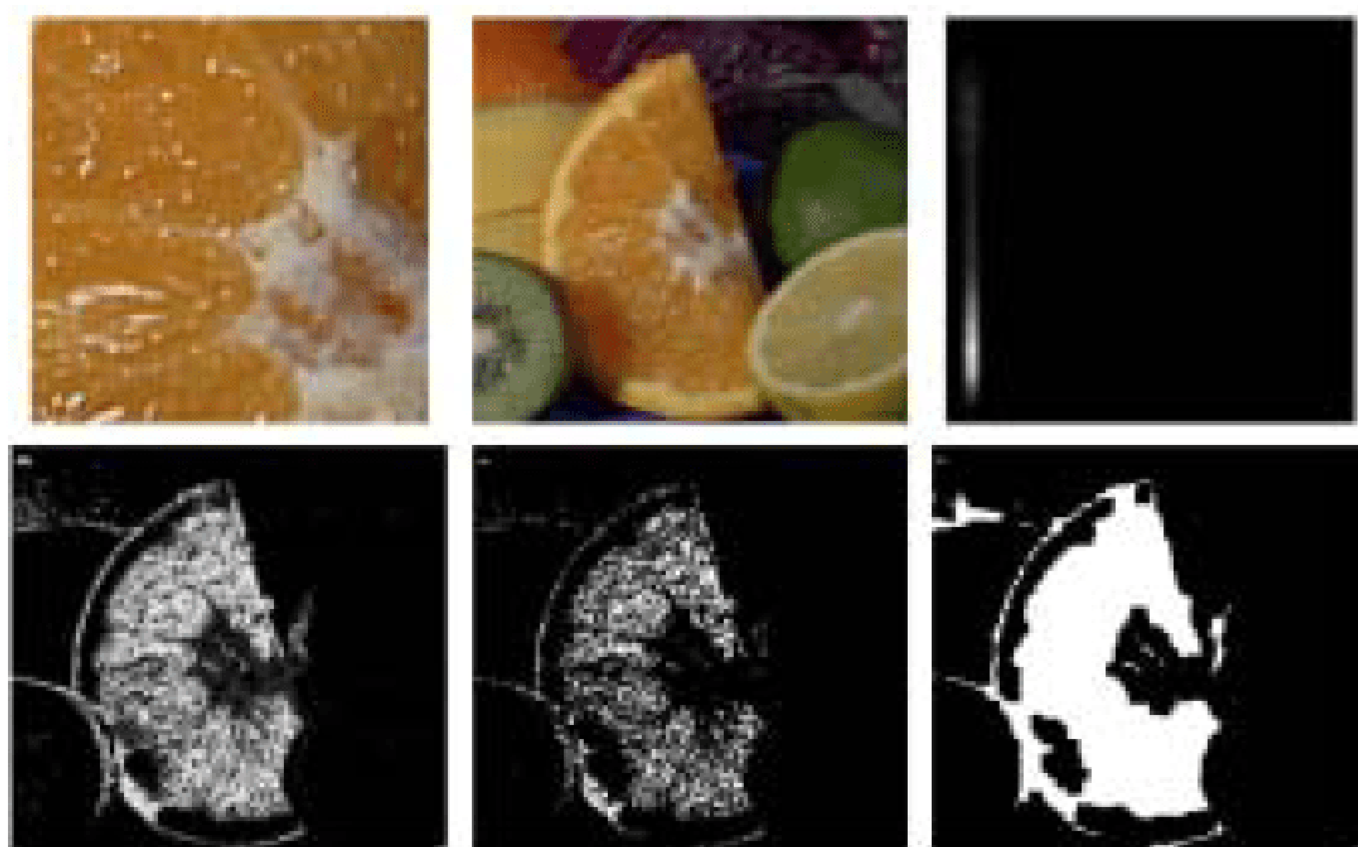


图 12-12　使用直方图的 Back Project 寻找图像中的橙子部分

```
import pyopencv as cv
import numpy as np

img = cv.imread("fruits_section.jpg") ❶
img_hsv = cv.Mat()
cv.cvtColor(img, img_hsv, cv.CV_BGR2HSV)
```

```
channels = cv.vector_int([0, 1])
result = cv.MatND()

r = cv.vector_float32([0, 256])
ranges = cv.vector_vector_float32([r, r])

cv.calcHist(cv.vector_Mat([img_hsv]), channels, cv.Mat(),  ❷
            result, cv.vector_int([40, 40]), ranges)

result[:] /= np.max(result[:]) / 255 ❸
```

❶首先载入颜色匹配的模板图像，并通过 cvtColor()将图像中 3 个通道的数据从蓝、绿、红变换为色相、饱和度和明度。cvtColor()可以在多种颜色表示法之间进行转换，通过第 3 个参数指定颜色的转换算法，具体的候选项请读者查看 OpenCV 的 API 文档。❷调用 calcHist()对模板图像的色相和饱和度这两个通道进行二维直方图计算。在色相和饱和度空间进行颜色匹配，能够得到较好的匹配结果。❸为了后续的 calcBackProject()计算不越界，这里将直方图的最大值缩小到 255。图 12-12(上右图)为所计算的直方图。

```
img2 = cv.imread("fruits.jpg") ❹
img_hsv2 = cv.Mat()
cv.cvtColor(img2, img_hsv2, cv.CV_BGR2HSV)

img_bp = cv.Mat()
cv.calcBackProject(cv.vector_Mat([img_hsv2]), ❺
                   channels=channels,
                   hist=result,
                   backProject=img_bp,
                   ranges = ranges)
```

❹载入目标图像，然后通过 cvtColor()进行颜色转换。❺调用 calcBackProject()将目标图像中每个像素的颜色变换为它在直方图中对应的值。它的第一个参数是一个图像 Vector 对象，hist 参数指定直方图，backProject 参数指定计算结果，它是一个单通道的形状和数值类型与输入图像相同的图像。channels、ranges 参数和 calcHist()的相同。图 12-12(下左图)为 calcBackProject()的计算结果。

```
img_th = cv.Mat()
cv.threshold(img_bp, img_th, 180, 255, cv.THRESH_BINARY) ❻
```

❻调用 threshold()对 calcBackProject()的输出图像进行二值化处理，参数 THRESH_BINARY 指定了二值化处理的方法，它将图像中值小于等于 180 的点都设置为 0，大于 180 的设置为 255。请读者查看 API 文档以了解更多的二值化方法。图 12-12(下中图)为二值化的结果。

```
struct = np.ones((3,3), np.uint8)
struct_mat = cv.asMat(struct, force_single_channel=True)
```

```
img_mp = cv.Mat()
cv.morphologyEx(img_th, img_mp, cv.MORPH_CLOSE, struct_mat, iterations=5) ❼
```

❼最后对二值化之后的图像进行形态学图像处理。使用 5 次开运算，将图像中分散的区域连接成一个大区域。图 12-12(下右图)为最终结果，它的白色区域正好对应目标图像中橙子的部分。我们还可以对这个结果进行一些处理：例如使用闭运算消除一些杂点，然后找出图像中面积最大的区域。

这种方法也可以用于在视频中跟踪一个颜色鲜明的物体。在跟踪物体之前，首先对一幅物体充满整个画面的图像进行直方图统计，然后对视频后续的帧使用 calcBackProject()进行计算。

12.3.4　二维离散傅立叶变换

图像数据可以看做一个二维离散信号，对其进行二维离散傅立叶变换，能将其转换为频域信号，将原始图像分解为众多二维正弦波的叠加。由于 NumPy 已经提供了二维离散傅立叶变换的函数，因此本节主要使用 NumPy 的相关函数进行说明。



为了更好地理解本节介绍的内容，需要读者掌握离散傅立叶变换相关的知识。在实战篇的第 15 章有一维离散傅立叶变换的详细论述。

我们知道，对一维的 N 点实数信号 x 进行快速傅立叶变换(FFT)之后，可得到表示频域信号的 N 个复数的数组 X。但是数据的信息量并没有增加，这是因为：

- 下标为 0 和 N/2 的两个复数的虚数部分为 0。
- 下标为 i 和 N-i 的两个复数共轭，也就是其虚数部分数值相同、符号相反。

同样，对于一个 N*N 的二维实数信号 x 进行二维快速傅立叶变换之后，可得到表示频域信号的 N*N 个复数元素的数组 X。其中，X[i,j]和 X[N-i,N-j]共轭，并且 X[0, 0]、X[0, N/2]、X[N/2, 0]、X[N/2, N/2]4 个元素的虚部为 0。下面我们用程序验证一下：

```
>>> import numpy as np
>>> from numpy import fft
>>> a = np.random.rand(8,8)
>>> b = fft.fft2(a) #将空域信号 a 转换为频域信号 b
```

首先载入相关的库，并且创建一个 8*8 的随机数数组 a，通过 fft2()得到表示频域信号的数组 b。让我们验证一下 b 的对称性：

```
>>> b[2,3], b[8-2,8-3] # 应该是共轭复数
((-0.70691285658224157+1.5149990601921564j),
 (-0.7069128565822419-1.5149990601921568j))
>>> np.allclose(b[1:,1:], b[7:0:-1,7:0:-1].conj())
True
```

通过数组的 conj()方法可以获得其共轭数组。由于存在计算误差，我们使用 allclose()比较两个数组的对应元素是否都足够相似。下面看看那 4 个虚部为 0 的元素：

```
>>> b[::4,::4]
array([[ 31.95957638+0.j,   1.85667587+0.j],
       [ -4.64227716+0.j,  -2.31274175+0.j]])
```

频域信号通过 ifft2()可以转换回空域信号，其结果和原始的空域信号完全相等。但是 ifft()得到的仍然是一个复数数组，只是每个元素的虚部都十分接近于 0：

```
>>> c = fft.ifft2(b) # 将频域信号转换回空域信号
>>> a[1,2], c[1,2]
(0.72192897413988666, (0.72192897413988677-1.1091723119464234e-16j))
>>> np.allclose(a, c) # 和原始信号进行比较
True
```

频域信号中的每个元素都对应空域信号中的一个二维正弦波，如果只选择将频域信号中的一部分转换回空域信号，那么就相当于对空域信号进行了滤波处理。下面的程序演示了将频域信号中不同区域转换回空域信号之后的滤波效果，其输出如图 12-13 所示。

opencv_numpy_fft2d.py
演示频域信号中各个区域对应的空域信号

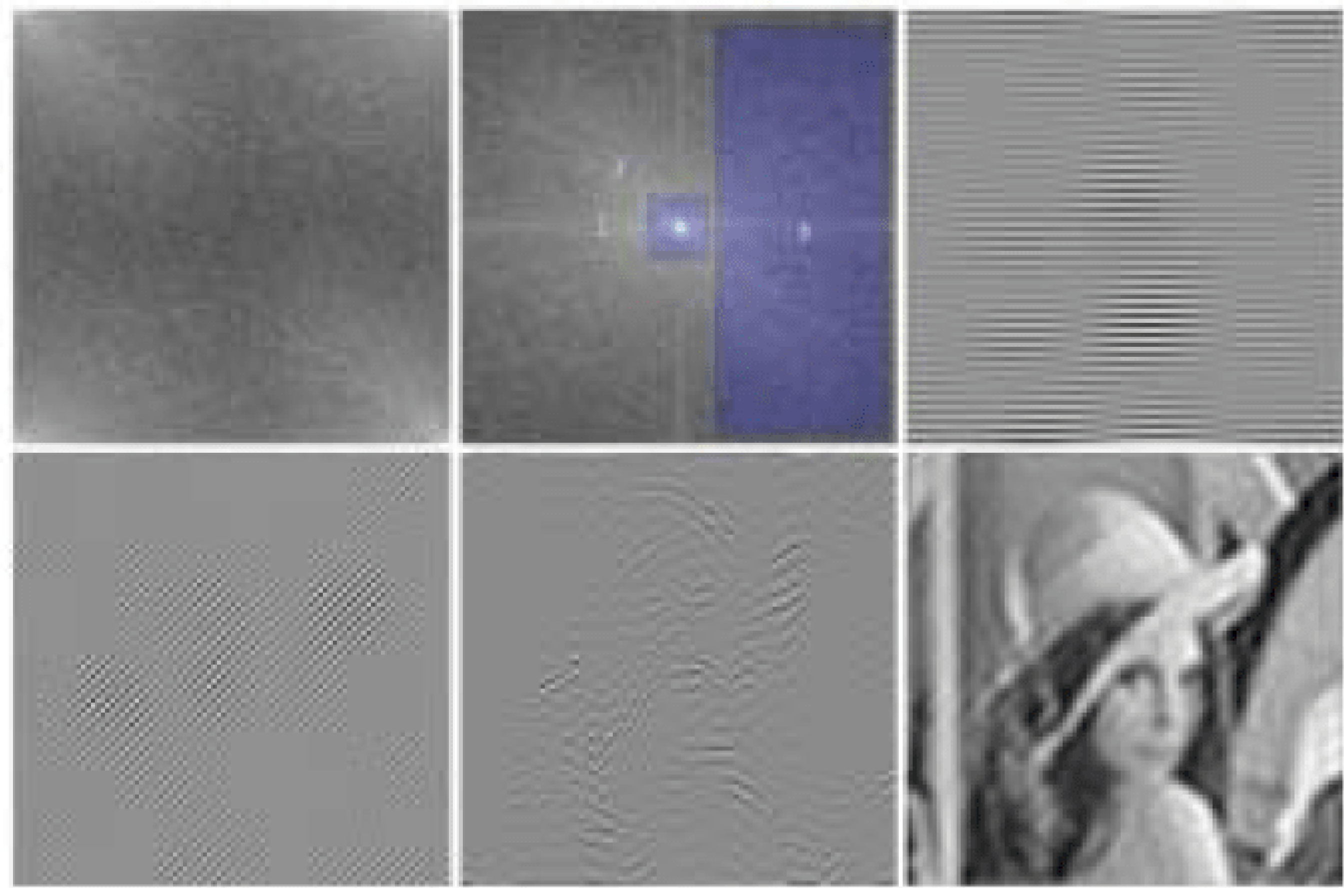


图 12-13 (左上)用 fft2()计算的频域信号，(中上)使用 fftshift()移位之后的频域信号，(其他)各个区域对应的空域信号

下面分析此程序中进行滤波部分的代码：

```
N = 256
```

```
img = cv.imread("lena_full.jpg")
img2 = cv.Mat()
cv.cvtColor(img, img2, cv.CV_BGR2GRAY)
img = cv.Mat()
cv.resize(img2, img, cv.Size(N, N))
```

首先载入一幅彩色图像，并将其转换为灰度图像。由于 FFT 运算的最佳大小为 2 的整数次幂，因此使用 resize()将图像的大小改为 256*256。

```
fimg = fft.fft2(img[:])
mag_img = np.log10(np.abs(fimg))
shift_mag_img = fft.fftshift(mag_img)
```

然后计算图像 img 的频域信号 fimg，由于它是一个复数数组，为了能将其作为图像显示，我们计算它的每个元素的模值，并取对数，得到数组 mag_img。图 12-13(左上)为 mag_img 的显示效果。模值图像的 4 个角与低频信号对应，中心与高频信号对应。由于 4 个角附近较亮，这说明原始图像的低频成分较多，这符合一般图像信号的规律。

为了更好地观察频域信号，我们使用 fftshift()对 mag_img 进行移位，得到数组 shift_mag_img。图 12-13(中上)为移位之后的模值图像。fftshift()将两个对角线上的方块对调，即 1、3 象限对调，2、4 象限对调。这样一来，图像的中部与低频对应，4 个角与高频信号对应。

```
rects = [(80,125,85,130),(90,90,95,95),
         (150, 10, 250, 250), (110, 110, 146, 146)]

filtered_results = []
for i, (x0, y0, x1, y1) in enumerate(rects):
    mask = np.zeros((N, N), dtype=np.bool) ❶
    mask[x0:x1+1, y0:y1+1] = True ❷
    mask[N-x1:N-x0+1, N-y1:N-y0+1] = True ❸
    mask = fft.fftshift(mask) ❹
    fimg2 = fimg * mask  ❺
    filtered_img = fft.ifft2(fimg2).real ❻
    filtered_results.append(filtered_img)
```

最后，我们选择频域信号中的一部分，将其转换回空域信号。图 12-13 的右上图和下排的图，分别显示中上图中 4 个矩形区域所对应的空域图像。

❶mask 是一个布尔数组，其形状和频域信号数组一样。❷将其中坐标在指定矩形范围之内的元素设置为 True。❸我们需要同时选择共轭对称的部分，否则通过 ifft2()转换回空域信号时虚部将不会为 0。❹通过 fftshift()对 mask 数组进行移位，使得它和频域信号 fimg 匹配。

❺接下来将频域信号 fimg 与 mask 相乘，得到在频域进行滤波之后的频域信号 fimg2。❻然后调用 ifft2()将 fimg2 转换回空域信号。

为了帮助读者理解频域信号和空域信号之间的关系，本书为读者提供了一个实时演示程

序，其界面如图 12-14 所示(见文前彩插)。在左边的频域模值图像上用鼠标绘制多边形区域，按住 Shift 键可以绘制多个区域。中间的图像显示根据用户所绘制的多边形计算的遮罩数组。右边的图像为频域信号经过遮罩处理之后所转换成的空域信号。

程序采用 TraitsUI 制作界面，使用 Chaco 绘图库进行快速绘图，使用 OpenCV 的 fillPoly()在遮罩数组上绘制多边形填充区域。程序中已经给出了较为详细的注释，请感兴趣的读者自行研究，这里就不再进行说明了。

opencv_fft2d_demo.py
实时演示图像的频域滤波



图 12-14　实时演示频域滤波，(左)频域的模值图像，(中)频域遮罩图像，(右)滤波之后的空域图像

12.4　图 像 识 别

OpenCV 除了能够对图像进行各种处理和变换之外，它还提供了大量的图像识别函数。

12.4.1　用霍夫变换检测直线和圆

用霍夫变换(Hough Transform)能够找出图像中的直线和圆。OpenCV 提供了如下 3 种霍夫变换相关的函数：

- HoughLines()：检测图像中的直线。
- HoughLinesP()：检测图像中的直线段。
- HoughCircles()：检测图像中的圆。

下面的演示程序使用 HoughLinesP()和 HoughCircles()进行线段和圆的检测。它们的各种参数均可以在控制面板中进行调整，运行界面如图 12-15 所示(见文前彩插)。

opencv_hough_demo.py
用霍夫变换寻找图像中的直线和圆

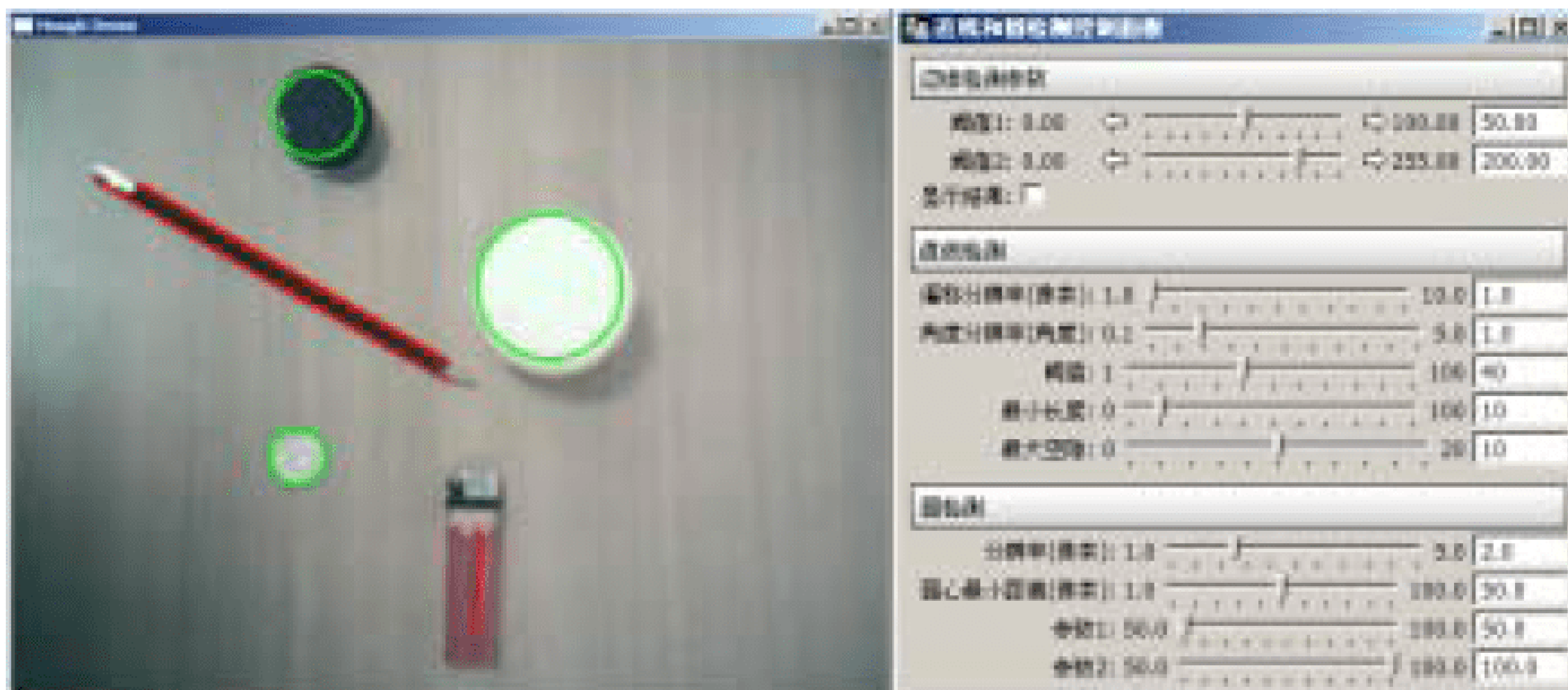


图 12-15　用霍夫变换寻找图像中的直线和圆

由于程序较长，这里仅对和图像检测相关的代码进行说明：

```
self.img = cv.imread("stuff.jpg")
self.img_gray = cv.Mat()
cv.cvtColor(self.img, self.img_gray, cv.CV_BGR2GRAY)
self.img_smooth = self.img_gray.clone()
cv.smooth(self.img_gray, self.img_smooth, cv.CV_GAUSSIAN, 7, 7, 0, 0)
```

首先在__init__()中初始化了 3 个属性——img、img_gray 和 img_smooth。img 是一幅 RGB 图像，使用 cvtColor()将其转换为灰度图像 img_gray。然后使用 smooth()对灰度图像进行高斯模糊，得到 img_smooth。后面的代码将对 img_gray 进行直线检测，对 img_smooth 进行圆形检测。

界面中的所有调节控件都分别和一个 Trait 属性相对应，当属性值发生变化时将调用 redraw()方法，按照最新的参数设置进行检测，并显示检测结果。下面介绍 redraw()方法中的代码：

```
edge_img = cv.Mat()
# 边缘检测
cv.Canny(self.img_gray, edge_img, self.th1, self.th2)
```

由于 HoughLinesP()需要针对二值图像进行操作，因此我们先用 Canny()对灰度图像进行边缘检测，得到一幅二值图像 edge_img。Canny()有两个阈值参数，它们直接影响边缘检测的结果。阈值越小，从图像中检测出来的边缘细节越多。



http://zh.wikipedia.org/zh-cn/Canny 算子
维基百科关于 Canny 算子的说明

```
# 计算结果
if self.show_canny:
    show_img = cv.Mat()
```

```
        cv.cvtColor(edge_img, show_img, cv.CV_GRAY2BGR)
    else:
        show_img = self.img.clone()
```

为了观察 Canny()的计算结果，在界面中有一个名为“显示结果”的复选框，与之对应的 Trait 属性为 show_canny。为了在图像中使用不同颜色绘制直线和圆，我们需要一幅彩色图像来保存最终的显示效果。因此，当 show_canny 为 True 时，使用 cvtColor()将边缘检测结果 edge_img 转换为彩色图像 show_img。当 show_canny 为 False 时，将原始图像复制一份并存储到 show_img 中。

```
    # 线段检测
    theta = self.theta / 180.0 * np.pi
    lines = cv.HoughLinesP(edge_img, ❶
        self.rho, theta, self.hough_th, self.minlen, self.maxgap)
    for line in lines: ❷
        cv.line(show_img,
            cv.asPoint(line[:2]),  ❸
            cv.asPoint(line[2:]),
            cv.CV_RGB(255, 0, 0), 2)
```

❶接下来调用 HoughLinesP()对边缘检测之后的二值图像进行线段检测。HoughLinesP()返回一个 vector_Vec4i 对象 lines，它是一个 Vec4i 对象的序列，其中每个 Vec4i 对象保存线段的两个端点的坐标。❷对 lines 中的每个 Vec4i 对象进行迭代，用 line()绘制线段，❸用 asPoint()创建表示线段两个端点的 Point 对象。在 Vec4i 对象中，前两个值表示线段的起点坐标，后两个值表示线段的终点坐标。

为了了解参数的含义，我们先学习一下直线霍夫变换的原理。

图像中的每条直线都可以用方程 $y=kx+m$ 表示。由于参数 k 和直线与 X 轴的夹角之间并不是线性关系，因此我们将直线方程改写为——以直线到原点的距离 r 和直线与 X 轴的夹角 θ 为参数，如图 12-16 所示：

$$y=\left(-\frac{\cos\theta}{\sin\theta}\right)x+\left(\frac{r}{\sin\theta}\right)$$

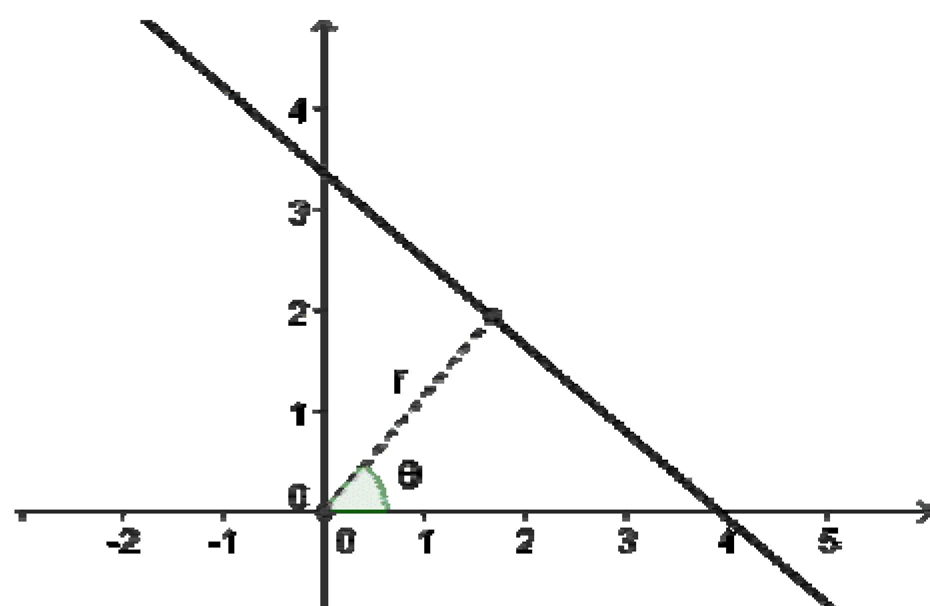


图 12-16　用 r 和 θ 表示的直线

经过图像中某个白色的点(x_0, y_0)的直线参数r和θ满足如下关系：

$$r = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta$$

它是一条在$\theta - r$空间中的正弦曲线。所谓霍夫变换，是指对于原始图像中的每个白色点(x_0, y_0)，绘制它们在$\theta - r$空间中对应的正弦曲线。众多正弦曲线的相交点(θ_0, r_0)就是原始图像中的一条直线。图 12-17 是一个简单的例子。其中，左图中的 4 个圆点构成一条直线，右图中与它们对应的正弦曲线(图中的实线)相交于一点，而与三角点对应的正弦曲线(虚线)则不经过此点(见文前彩插)。

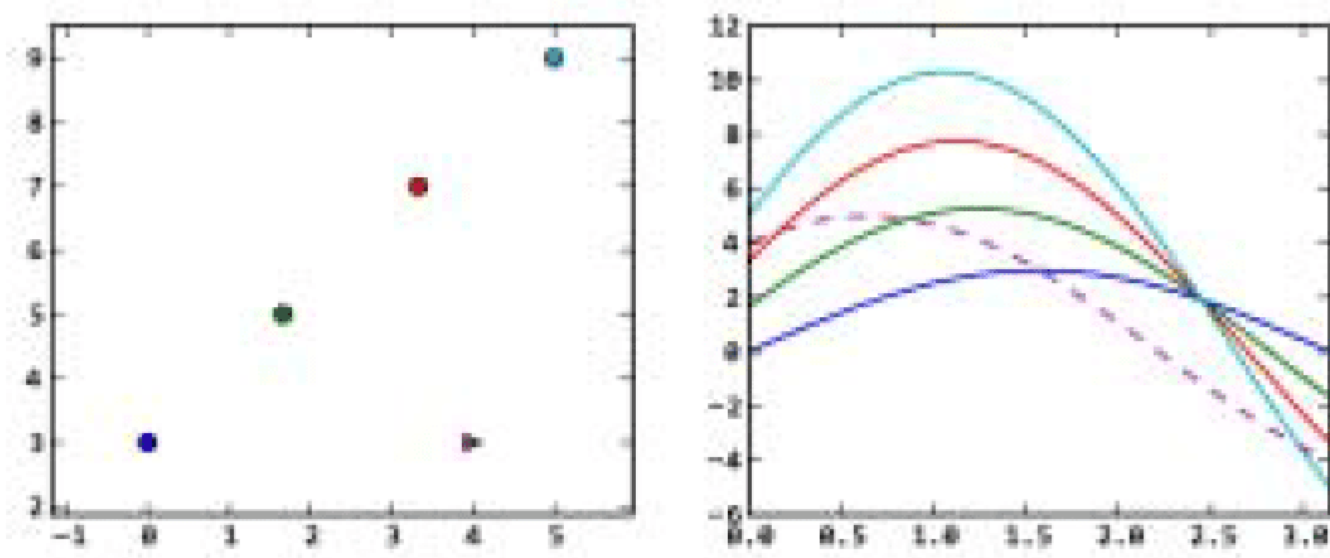


图 12-17　霍夫变换示意图

在实际计算时，我们使用一个表示$\theta - r$空间的灰度图像作为累加器，其中每个点对经过此点的正弦曲线进行计数。然后通过阈值找出累加器中所有的峰值点，这些峰值点对应的$\theta - r$坐标就是原始图像中的直线参数。

HoughLinesP()的参数如下：

```
HoughLinesP( image, rho, theta, threshold, minLineLength=0, maxLineGap=0)
```

其中，image 参数为进行直线检测的图像，rho 和 theta 参数分别为累加器中每个点所表示的r和θ的大小。其中，rho 的单位是像素点，theta 是以弧度表示的角度。值越小，累加器的尺寸越大，最后寻找出的直线的参数精度越高，但是运算时间也越长。threshold 参数是在累加器中寻找峰值时所使用的阈值，即只有大于此值的峰值点才被认为与某条直线对应。由于 HoughLinesP()检测的是图像中的线段，因此 minLineLength 参数指定线段的最小长度，maxLineGap 参数指定线段的最大间隙。当有多条线段共线时，间隙小于此值的线段将被合并为一条线段。

```
# 圆形检测
circles = cv.HoughCircles(self.img_smooth, 3,  ❹
    self.dp, self.mindist, param1=self.param1, param2=self.param2)
for circle in circles:
    cv.circle(show_img,
        cv.Point(int(circle[0]), int(circle[1])), int(circle[2]),  ❺
        cv.CV_RGB(0, 255, 0), 2)
```

❹接下来寻找圆形。由于 HoughCircles()内部会对图像进行边缘检测，因此这里使用灰度图像。为了减小噪声影响，我们使用模糊之后的图像。HoughCircles()的参数如下：

```
HoughCircles(image, method, dp, minDist, param1=100, param2=100, minRadius=0,
maxRadius=0)
```

其中，method 参数为圆形检测的算法，目前 OpenCV 中只实现了一种检测算法——CV_HOUGH_GRADIENT。由于 PyOpenCV 中没有定义此常数，因此我们直接使用它所对应的整数值 3。dp 参数和直线检测中的 rho 参数类似，决定了检测的精度，dp=1 时累加器的分辨率和输入图像相同，dp=2 时累加器的分辨率为输入图像的一半。minDist 参数是检测到的所有圆的圆心之间的最小距离，当它过小时会检测出很多近似的圆形，过大则可能会漏掉一些结果。

param1 和 param2 参数是和检测算法相关的参数。其中，param1 参数相当于边缘检测 Canny()的第二个阈值，Canny()的第一个阈值自动设置为它的一半。param2 参数是累加器上的阈值，它的值越小，检测出的圆形越多。minRadius 和 maxRadius 参数指定圆形的半径范围，默认都为 0，表示范围不限。

❺HoughCircles()返回一个 vector_Vec3f 对象 circles，它是一个 Vec3f 对象的序列，其中的每个 Vec3f 对象保存圆心坐标和半径。由于 Vec3f 对象中的数值为浮点数，因此在调用 circle()绘制圆形时，需要将它们转换成整数。

由于圆形有 3 个参数——圆心 X 坐标、圆心 Y 坐标和半径，如果直接使用三维累加器，计算效率太低。并且由于累加器中每个点的累计次数不能足够多，会出现很多局部峰值，这也会影响检测结果。因此，HoughCircles()使用了一种被称为霍夫梯度的算法来进行圆形检测。它的计算步骤如下：

(1) 首先将原始图像经过边缘检测算法获得一张边缘图像，这里使用 Canny()进行边缘检测，并使用 param1 参数指定其阈值。

(2) 对于边缘图像中每个白色的点(x0, y0)，计算其局部梯度，这里使用 Sobel()进行梯度计算。假设白色点为圆周上的某点，经过(x0, y0)且沿着梯度方向的直线将通过圆心。

(3) 对于梯度直线上离点(x0, y0)的距离在 minRadius 和 maxRadius 之间的所有点，在累加器中进行计数。

(4) 累加器中大于阈值 param2 的局部峰值为图像中所检测出的圆形的中心。

(5) 对于每个检测出的圆心，在边缘图像中寻找离它距离相同的白色点的集合，并计算出半径。如果此圆心有足够多的白色点支持，那么它就是一个真正的圆心。

12.4.2 图像分割

一般的图像颜色丰富、信息繁杂，不利于计算机进行图像识别。因此通常会使用图像分割技术，将图像中相似的区域进行合并，使得图像更容易理解和分析。本节将介绍 OpenCV 中提供的 3 种常见的图像分割算法。

1. 图像金字塔算法

所谓图像金字塔，就是将原始图像按比例进行放大或缩小，得到一系列不同分辨率的图像。pyrSegmentation()利用图像金字塔，先在低分辨率的图像中进行图像分割，得到一个初始的分割集合，然后随着分辨率的提高，逐渐调整。它的调用参数如下：

```
pyrSegmentation(src, dst, storage, level, threshold1, threshold2)
```

其中，src 参数是需要进行分割处理的图像，dst 参数是进行图像分割处理后的结果。和以前介绍的图像处理函数不同，这里需要传递一个事先分配好内存的图像对象，因此不能使用空图像。storage 参数是一个 MemStorage 对象，使用 createMemStorage()创建，在其中保存分割之后的各个区域的信息。

level 参数是一个整数，表示图像金字塔的层数，根据图像大小的不同，所能设置的最大层数也会发生变化。由于金字塔两层之间的大小比例为 2，因此 pyrSegmentation()只能处理宽和高为 2 的整数倍的图像。如果图像尺寸不符合此条件，可以事先对图像进行缩放或裁剪。

threshold1 和 threshold2 参数是两个用来控制区域合并的阈值，当两个像素或区域的颜色差别小于阈值时将进行合并，关于其具体含义请读者参考 API 文档。pyrSegmentation()返回一个 CvSeq 对象，保存各个分割区域的信息。目前的 PyOpenCV 版本还没有提供方法来获取其中的内容。

下面的程序演示了 threshold2 参数对图像分割结果的影响，效果如图 12-18 所示。可以看到：阈值越大，越多的像素将被合并为同一种颜色(见文前彩插)。



opencv_pyrSegmentation.py
用 pyrSegmentation()进行图像分割

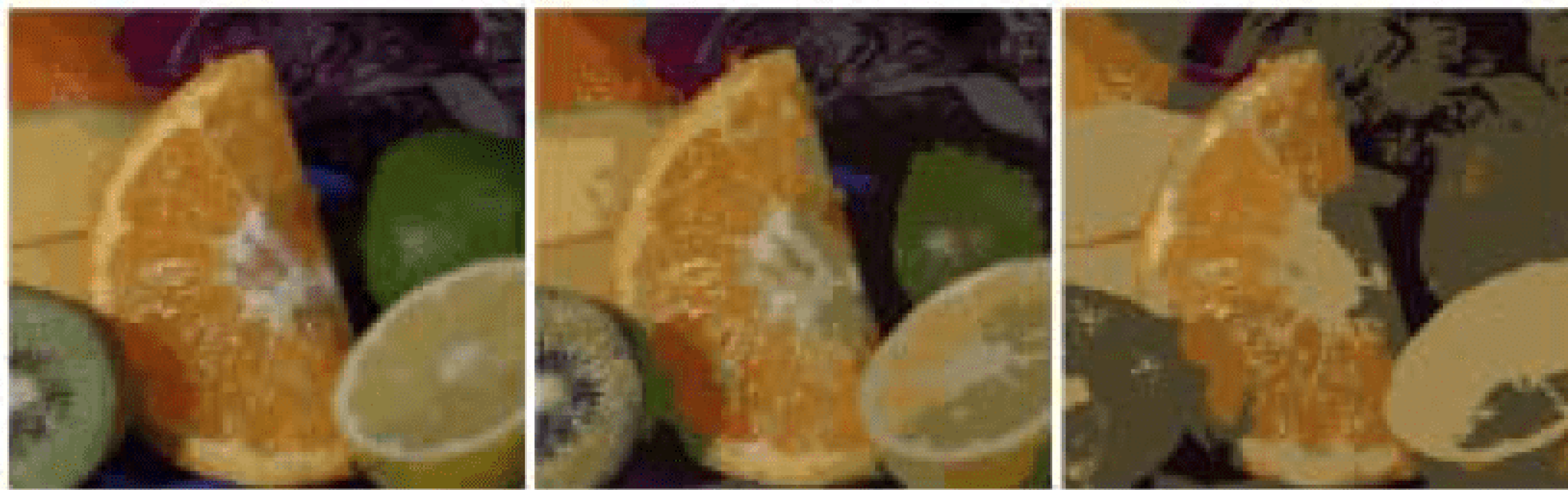


图 12-18　用 pyrSegmentation()进行图像分割，从左到右的阈值分别为 10、30、60

```
img = cv.imread("fruits.jpg")
threshold2 = [10, 30, 60]
for i, th2 in enumerate(threshold2):
    img2 = img.clone()
    storage = cv.createMemStorage(0)
    result = cv.pyrSegmentation(img, img2, storage, 4, 200, th2)
```

这里通过 img.clone()快速获得一个和 img 的大小和像素类型完全相同的图像 img2，其中的颜色信息不会被使用。

2. Mean-Shift 算法

pyrMeanShiftFiltering()使用 Mean-Shift 算法对图像进行分割，它的调用参数如下：

```
pyrMeanShiftFiltering(src, dst, sp, sr, max_level=1,
    termcrit=TermCriteria(type=3, maxCount=5, epsilon=1.0))
```

其中，src 参数是原始图像，dst 参数是目标图像。和 pyrSegmentation()一样，dst 必须是一个已经分配好内存的图像，其大小和像素类型和 src 一致。

pyrMeanShiftFiltering()以 src 中的每个点(x,y)为初始点，寻找与它邻近的点。这里的邻近点必须满足下面两个条件：

- 在以(x,y)为中心的边长为 2*sp 的正方形范围内。
- 和点(x,y)的各个通道值的差的平方和小于 sr 参数，也就是将 3 个颜色通道当做三维向量，在此颜色空间中，两个点的距离小于 sr。

然后计算邻近点的坐标平均值和颜色平均值，并据此平均点再次寻找图像中的邻近点。如此迭代下去，直到满足迭代终止条件。将迭代终止时的颜色平均值写进图像 dst 中的坐标点(x,y)。

在 OpenCV 中，所有的迭代算法都可以用 TermCriteria 对象设置迭代相关的参数。TermCriteria 对象有三个属性：maxCount 为最大迭代次数；epsilon 为迭代终止时的误差，即两次迭代结果的差小于此值时将结束计算；type 指定哪种终止条件有效，3 表示两种终止条件都有效。

可以通过 max_level 参数指定使用图像金字塔进行计算。当使用图像金字塔时，它先对低分辨率的图像进行分割计算，然后利用此结果对高分辨率的图像进行分割。

下面是使用 pyrMeanShiftFiltering()进行图像分割的演示程序。由于程序比较简单，为了节省篇幅，这里就不列出源代码了，效果如图 12-19 所示(见文前彩插)。



opencv_pyrMeanShiftFiltering.py, opencv_pyrMeanShiftFiltering_demo.py
使用 pyrMeanShiftFiltering()进行图像分割



图 12-19　使用 pyrMeanShiftFiltering()进行图像分割，从左到右参数 sr 分别为 20、40、80

3. 分水岭算法

分水岭(Watershed)算法的基本思想是将图像的梯度当作一个地形图。图像中变化小的区域相当于地形图中的山谷，而变化大的区域相当于山峰。从指定的几个初始区域同时开始向地形灌不同颜色的水[②]，水面上升逐渐淹没山谷，并且范围逐渐扩大。当所有区域的水面连接到一起时，所得到的不同颜色的灌溉区域就是最终的图像分割结果，最终的分割区域数和初始区域数相同。

在 OpenCV 中，使用 watershed()实现此算法，它的调用形式如下：

```
watershed(image, markers)
```

image 参数是需要进行分割处理的图像，它必须是一个 3 通道的 8 位图像。markers 参数是一个单通道的 32 位浮点数图像，其大小必须和 image 相同。markers 中值大于 0 的点构成初始灌溉区域，其值可以理解为水的颜色。调用 watershed()之后，markers 中几乎所有的点都将被赋值为某个初始区域的值，而在两个区域边界线上的点将被赋值为-1。

我们使用下面的交互式演示程序帮助用户观察初始区域和最终分割结果之间的关系。执行此程序之后，将显示一幅水果照片。用鼠标左键在图像上绘制初始区域，每次都将使用不同颜色。每次松开左键时，将使用目前的初始区域进行图像分割，并用颜色显示分割之后的各个区域，效果如图 12-20 所示(见文前彩插)。当有两个初始区域时，整个图像被分割为两块：绿色标出的猕猴桃区域和所有其他的区域。当我们逐渐增加初始区域时，图像中各个水果的边线逐渐被寻找出来。



opencv_watershed_demo.py
用 watershed()进行图像分割

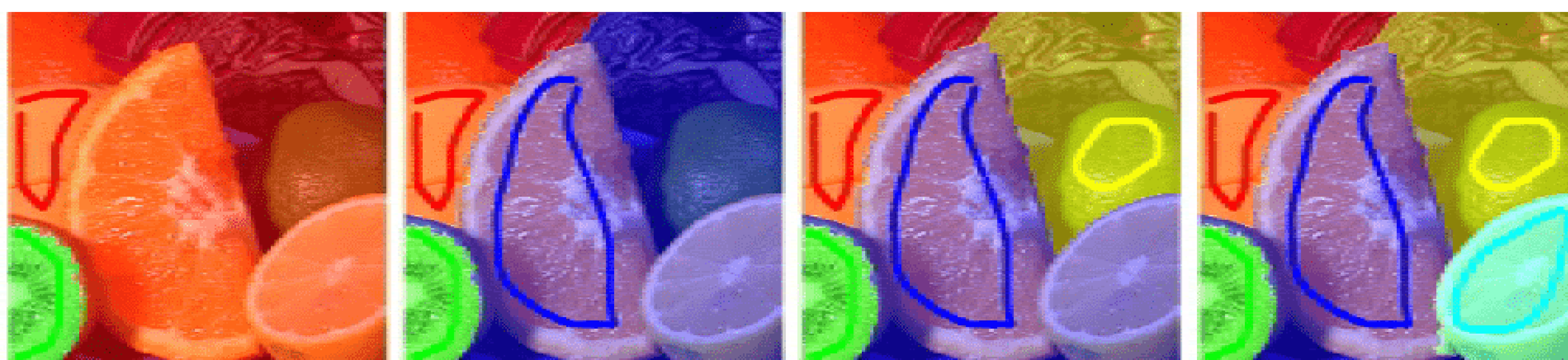


图 12-20 watershed()的图像分割结果，从左往右逐步添加新的初始区域

下面分析一下程序：

```
# 区域的颜色列表
marks_color = [ ❶
    cv.CV_RGB(0, 0, 0)    ,cv.CV_RGB(255, 0, 0),
    cv.CV_RGB(0, 255, 0)  ,cv.CV_RGB(0, 0, 255),
```

② 这里所说的“颜色”是用来区分不同区域的一个数值，和图像的颜色没有任何关系。

```
    cv.CV_RGB(255, 255, 0),cv.CV_RGB(0, 255, 255),
    cv.CV_RGB(255, 0, 255),cv.CV_RGB(255, 255, 255)
]

# 将颜色列表转换为调色板数组，只取前 3 个通道的值
palette = np.array([c.ndarray[:-1] for c in marks_color], dtype=np.uint8) ❷

seed = 1 # 从序号 1 开始设置区域颜色
mask_opacity = 0.5 # 绘制区域颜色的透明度
```

❶首先，列表 marks_color 定义了各个区域的颜色，颜色的下标用于设置 markers 数组，而颜色的值将用于绘制的效果图。注意，这里的第一个黑色不起作用，因为我们将从 1 开始设置 markers 数组。❷将颜色列表转换成一个二维数组 palette，此数组在将分割区域转换回图像时作为调色板使用。注意，我们只使用前 3 个通道的值，因此 palette 数组的形状为(8, 3)。

```
img = cv.imread("fruits.jpg")
img2 = img.clone() # 绘制初始区域用
markers = cv.Mat(img2.size(), cv.CV_32S) # 存储初始区域的数组
markers[:] = 0

cv.namedWindow("Watershed Demo")
cv.imshow("Watershed Demo", img2)
cv.setMouseCallback("Watershed Demo", mouse_call_back) ❸
cv.waitKey(0)
```

首先载入图像，并且复制一份为 img2，它将被用于绘制初始区域。然后创建 markers 数组，并将其初始化为 0。❸为了处理鼠标事件，我们使用 setMouseCallback()将 mouse_call_back()函数设置为窗口"Watershed Demo"的鼠标事件处理函数。在窗口中发生的任何鼠标事件都将调用 mouse_call_back()进行处理。

```
def mouse_call_back(event, x, y, flags, user_data):
    global seed

    # 右键松开时，初始化种子图像
    if event == cv.CV_EVENT_RBUTTONUP: ❹
        img2[:] = img[:]
        markers[:] = 0
        seed = 1
        cv.imshow("Watershed Demo", img2)

    if seed == len(marks_color): return

    # 左键按下时，在种子图像上添加种子
    if flags == cv.CV_EVENT_FLAG_LBUTTON: ❺
        pt = cv.Point(x, y)
```

```
        cv.circle(markers, pt, 5, cv.Scalar(seed,seed,seed,seed), cv.CV_FILLED)
        cv.circle(img2, pt, 5, marks_color[seed], cv.CV_FILLED)
        cv.imshow("Watershed Demo", img2)

    # 左键松开时，使用 watershed 进行图像分割
    if event == cv.CV_EVENT_LBUTTONUP:
        seed += 1
        tmp_markers = markers.clone() ❻
        cv.watershed(img, tmp_markers)
        color_map = tmp_markers[:].astype(np.int) ❼

        img3 = img2.clone()
        img4 = cv.asMat( palette[color_map] ) ❽
        cv.addWeighted(img3, 1.0, img4, mask_opacity, 0, img3) ❾
        cv.imshow("Watershed Demo", img3)
```

鼠标事件处理函数的参数列表如下：

```
mouse_call_back(event, x, y, flags, user_data)
```

其中，event 参数为发生的鼠标事件，x 和 y 参数为此时鼠标的坐标，flags 参数是鼠标的按键状态。❹当 event 为 CV_EVENT_RBUTTONUP，即释放右键时，我们初始化 img2、markers 和 seed 等，并且刷新显示。❺当鼠标左键为按下状态时，以 5 个像素为半径分别在 markers 和 img2 上绘制填充圆。在 markers 上用 seed 作为颜色，在 img2 上用 seed 对应的颜色绘图。



当处理 OpenCV 图像窗口的鼠标事件时，需要执行 waitKey()函数。此时无法在同一个线程中同时运行 TraitsUI 界面。

当鼠标左键松开时，调用 watershed()进行图像分割，并绘制分割的结果。❻为了保证 markers 中的内容不被覆盖，我们将它的复本 tmp_markers 传递给 watershed()。分割完成之后，tmp_markers 中将保存图像分割的结果。❼由于它的元素为浮点数，因此先将其转换为整数数组 color_map，color_map 中的元素值为颜色的下标。❽使用前面创建的调色板数组 palette 将 color_map 转换成一个 3 通道的图像 img4。❾最后将 img4 以半透明的方式叠加到 img2 的复本 img3 上。这里使用 addWeighted()进行叠加计算，其调用参数如下：

```
addWeighted( a, alpha, b, beta, gamma, c)
```

其中，a、b、c 是 3 个大小和通道数都相同的数组，alpha、beta 和 gamma 是 3 个浮点数，addWeighted()完成如下计算，其中的 I 表示多维下标：

```
c[I] = a[I] * alpha + b[I] * beta + gamma
```

也可以使用 NumPy 进行同样的计算，但是 addWeighted()的好处在于它使用饱和加法，当结果超出 c 的数值范围时将不会出现翻转问题。例如，某点的计算结果为 257，如果使用 NumPy 计算，会将 1 保存到 c；而如果使用 addWeighted()，会对 257 进行饱和处理，将数值范围的最大值 255 保存到 c 中。

12.4.3 用 SURF 进行特征匹配

SURF[③]是一种快速提取图像的特征点的算法，它所提取的图像特征具有缩放和旋转的不变性，而且它对图像的灰度变化也不敏感。针对 SURF 算法的详细介绍已超出本书的范围，本节只简单地介绍 OpenCV 中 SURF 类的用法。下面的程序演示了如何在一幅图及其仿射变换结果图之间寻找匹配的特征点，运行效果如图 12-21 所示。

opencv_surf_demo.py
用 SURF 寻找图像之间匹配的特征点

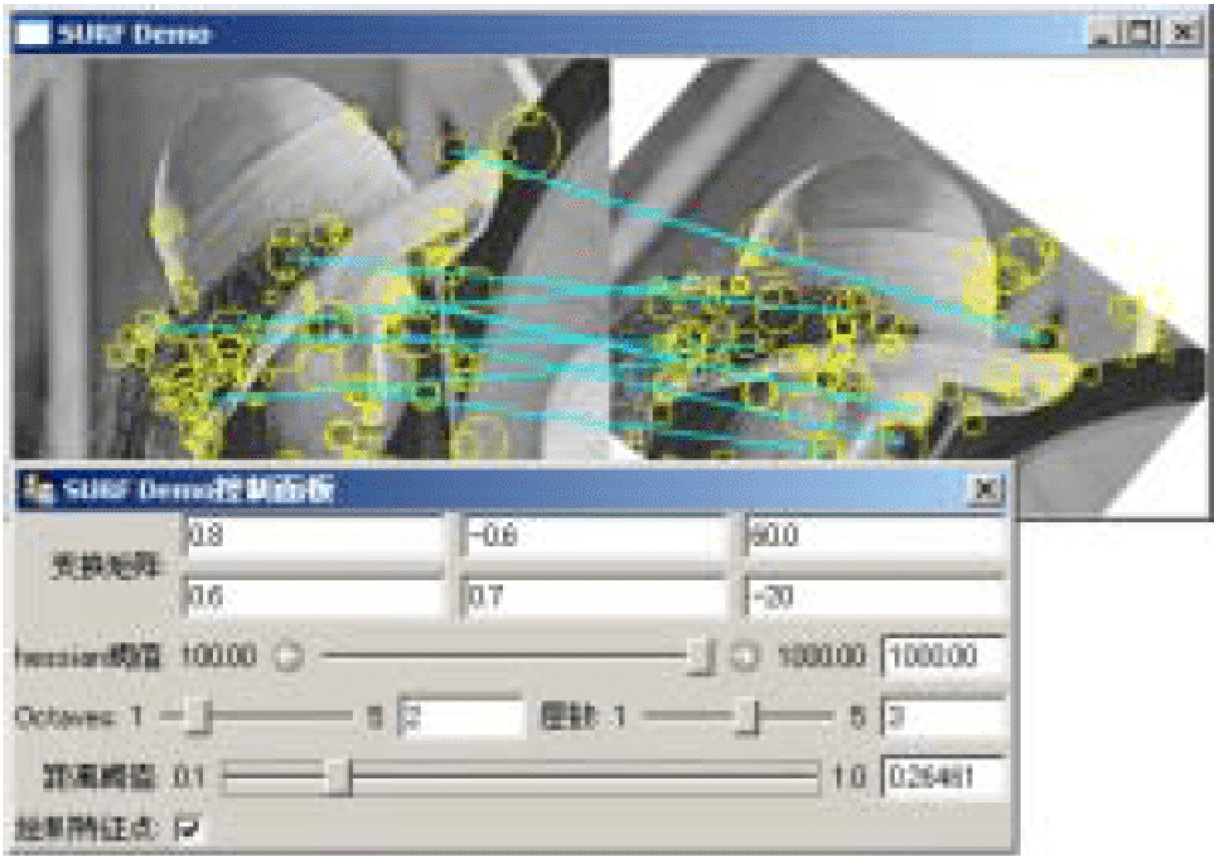


图 12-21　用 SURF 寻找图像之间匹配的特征点

界面中，通过“变换矩阵”设置仿射变换矩阵，而“hessian 阈值”、“Octaves”和“层数”则是 SURF 算法的 3 个参数。使用 SURF 得到的特征包括两个部分：描述特征位置、尺寸和方向的特征点对象，以及描述特征样式的特征向量。如果在界面中选中了“绘制特征点”复选框，在图像上将用圆形表示出特征点的位置和尺寸。每个特征点都有一个特征向量与其对应，通过计算两幅图的两组特征点之间所有特征向量的距离，可以找出特征相似的两组特征点。程序中用直线连接所有特征向量距离小于“距离阈值”的特征点。由于程序较长，这里只对程序中与特征点提取相关的部分进行说明：

```
def get_features(self, img):
```

③ 英文全称为 Speeded-Up Robust Features。

```
        surf = cv.SURF(self.hessian_th, self.octaves, self.layers, True)  ❶
        keypoints = cv.vector_KeyPoint()
        features = surf(img, cv.Mat(), keypoints)  ❷
        return keypoints, np.array(features)
```

首先，get_features()通过调用 SURF()寻找出图像的特征点。❶SURF 是一个类，它的初始化方法中有 4 个参数：

```
    SURF.__init__(self, _hessianThreshold ,_nOctaves, _nOctaveLayers, _extended=False)
```

其中，前 3 个参数分别与界面中的“hessian 阈值”、“Octaves”和“层数”相对应，请读者在演示界面中调节这些参数以了解它们是如何影响所提取的特征点的。参数_extended 决定了每个特征点的特征向量的长度，False 表示向量长度为 64，True 表示向量长度为 128。

直接调用 SURF 对象，即可对图像进行特征点提取，它有两种调用方式：

```
    SURF.__call__(self, img, mask)
```

img 参数是要提取特征点的图像，通过 mask 参数可以指定 img 的遮罩图像。它返回的是一个 KeyPoint 对象的列表，图像中每个特征点都由 KeyPoint 对象进行描述。读者可以在 IPython 中输入“cv.KeyPoint?”来查看它的文档说明。其中保存了特征点的位置、大小及方向等信息。此方法只能获得图像的特征点列表，而不能获得每个特征点的特征向量。下面的调用方法则可以同时获得特征点和特征向量：

```
    SURF.__call__(self, img, mask, keypoints, useProvidedKeypoints=False)
```

它返回的是一个 1 行 N 列的单精度浮点数数组，我们可以把它看做一维数组。其中顺序存储着与特征点列表中每个特征点对应的特征向量。当 useProvidedKeypoints 参数为 False 时，它会将特征点都添加到 keypoints 列表中。❷程序中使用这种调用方式同时获得特征点和特征矢量。

对两幅图进行特征提取之后，它们的特征点和特征向量分别保存在 keypoints1、features1、keypoints2、features2 等几个属性中。下面的代码计算 features1 和 features2 之间的距离：

```
    def match_features(self):
        f1 = self.features1.reshape(len(self.keypoints1), -1) ❸
        f2 = self.features2.reshape(len(self.keypoints2), -1)
        self.f1 = f1
        self.f2 = f2
        distances = cdist(f1, f2) ❹
        self.mindist = np.min(distances, axis=1)  ❺
        self.idx_mindist = np.argmin(distances, axis=1)
```

❸调用 reshape()方法将特征向量转换成二维数组。它的第 0 轴的长度为特征点的个数，第 1 轴的长度为特征向量的长度。❹然后调用 cdist()计算出距离矩阵 distances，cdist()是 SciPy 中

的一个特殊函数，因此在程序开头使用下面的语句载入它：

```
from scipy.spatial.distance import cdist
```

使用 cdist(f1, f2)可以快速计算输入的两组向量 f1 和 f2 中所有两对向量之间的距离，它返回一个形状为(f1.shape[0], f2.shape[0])的二维数组，其中每个值是与其对应的两个向量的距离。由于所有的计算都在 C 语言级别完成，因此运算速度很快。它相当于使用 NumPy 进行了如下计算[④]：

```
distances[i, j] = np.sqrt(np.sum((f1[i] - f2[j])**2))
```

❺接下来找出 distances 中每行的最小值以及最小值的下标，分别保存为 mindist 和 idx_mindist 属性。于是，idx_mindist[i]就是 f2 中与 f1[i]距离最近的向量的下标。

最后，在 redraw()方法中使用下面的语句，用直线连接两幅图像中距离小于阈值的特征点：

```
# 绘制直线，连接距离小于阈值的两个特征点
for idx1 in np.where(self.mindist < self.max_distance)[0]:  ❻
    idx2 = self.idx_mindist[idx1]
    pos1 = self.keypoints1[int(idx1)].pt  ❼
    pos2 = self.keypoints2[int(idx2)].pt
    p1 = cv.Point(int(pos1.x), int(pos1.y)) ❽
    p2 = cv.Point(int(pos2.x)+w, int(pos2.y))
    cv.line(show_img, p1, p2, cv.CV_RGB(0,255,255), lineType=16)
```

❻用 where()找到 mindist 中小于指定阈值的下标。❼对这些下标进行循环，通过 idx_mindist 获得右图中(参见图 12-21)与其距离最近的特征点的下标，然后通过 KeyPoint 对象的 pt 属性获得特征点的位置信息。❽最后将位置转换为 Point 对象，用来绘制线段。为了让线段的右端点和右图中的特征点重合，需要对右图特征点的横坐标加上左图的宽度 w。

④ 实际上，如果想用 NumPy 快速计算此距离矩阵，可以通过广播运算，在 Python 级别只使用一层循环实现。

第 13 章

数据和文件

有了数据才能够处理，因此让我们首先了解如何利用 Python 的扩展库读写各种各样的数据。本章介绍如何处理声音、动画、Excel 和 HDF5 等多种格式的文件，以及如何从声卡、摄像头等设备实时读取数据。

13.1 声音的输入输出

标准的 Python 库支持 WAV 文件的读写，而实时的声音输入输出则需要安装 pyAudio 模块。掌握了这些基础知识之后，就可以做许多有趣的声效处理实验了。声效处理方面的内容将在以后的章节介绍。

13.1.1 读写 WAV 文件

WAV 是微软公司开发的一种声音文件格式，虽然支持多种压缩格式，但它却经常被用来保存未压缩的声音数据。WAV 文件有 3 个重要的参数：声道数、取样频率和量化位数。

- 声道数：可以是单声道或双声道。
- 采样频率：一秒内对声音信号的采集次数，常用的有 8k Hz、16k Hz、32k Hz、48k Hz、11.025k Hz、22.05k Hz、44.1k Hz。
- 量化位数：一次采样所采集数据的比特数，通常有 8 bit、16 bit、24 bit 和 32 bit 等几种。

如果读者需要自己录制和编辑声音文件，推荐使用 Audacity，它是一款开源、跨平台、多声道的录音编辑软件。可以用它进行声音信号的录制，然后再输出为 WAV 文件供 Python 程序处理。



http://audacity.sourceforge.net
开源录音编辑软件 Audacity 的下载地址

下面让我们看看如何在 Python 中读写声音文件，下面的程序可绘制如图 13-1 所示的 Windows XP 的警告声波形：



read_wave.py
读取 WAV 文件并绘制波形

```
import wave
import pylab as pl
import numpy as np

# 读取格式和数据
f = wave.open(r"c:\WINDOWS\Media\ding.wav", "rb") ❶
nchannels, sampwidth, framerate, nframes = f.getparams()[:4] ❷
str_data = f.readframes(nframes) ❸
f.close()

#将波形数据转换为数组
wave_data = np.fromstring(str_data, dtype=np.short) ❹
wave_data.shape = -1, nchannels ❺
time = np.arange(0, nframes) * (1.0 / framerate) ❻

# 绘制波形
pl.subplot(211)
pl.plot(time, wave_data[:,0])
pl.subplot(212)
pl.plot(time, wave_data[:,1], c="g")
pl.xlabel("time (seconds)")
pl.show()
```

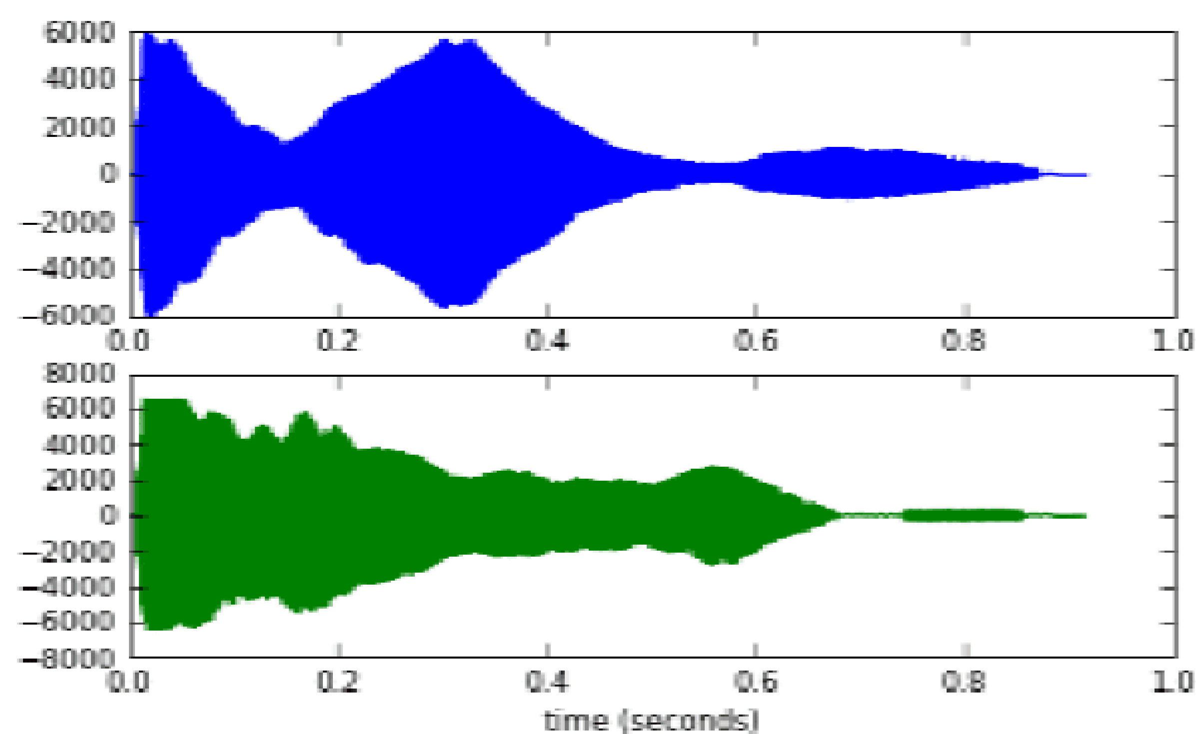


图 13-1　Windows XP 的经典“叮”声的波形

首先载入处理 WAV 文件的标准模块 wave，❶然后调用 wave.open()打开 WAV 文件，注意需要使用二进制模式"rb"打开文件。open()返回一个 Wave_read 对象，通过它的方法可以读取 WAV 文件的格式和数据。

❷调用 getparams()读取 WAV 文件的所有格式信息，它返回的是一个元组，其中的每个元素分别表示声道数、以字节为单位的量化位数、采样频率、采样点数、压缩类型、压缩类型的描述。wave 模块只支持未压缩的声音数据，因此可以忽略最后两个格式信息。这些信息也可

以分别用 getnchannels()、getsampwidth()、getframerate()、getnframes()获得。

❸调用 readframes()读取声音数据，其参数为需要读取的取样点数，这里传递整个文件的采样点数 nframes，表示读入整个 WAV 文件的数据。readframes()返回用字符串表示的二进制数据。

❹接下来需要根据声道数和量化单位，将读取的二进制数据转换为 NumPy 数组。调用 fromstring()将字符串转换为数组，dtype 参数指定数组的数据类型。由于我们的声音格式是以两个字节表示一个取样值，为了方便起见，这里直接设置 dtype 参数为 short 类型。更完美的实现需要根据量化位数 sampwidth 选择合适的数据类型。

现在得到的 wave_data 是一个一维短整型数组，但我们的声音文件是双声道的，它由左右两个声道的取样交替构成。因此❺设置 wave_data 数组的 shape 属性，修改其形状，这样一来，数组的第 0 轴的长度为取样点数，第 1 轴的长度为声道数。

❻为了绘图方便，通过取样点数和取样频率计算出每个取样的时刻。

上面的程序是将 WAV 文件中的数据一次性全部读入内存。如果希望读取声音文件中的某一段数据，可以先调用 wave_read 对象的 setpos(pos)方法，将文件指针移动到指定位置，其中的 pos 参数是跳过的取样点数。然后再调用 readframes()读取指定长度的声音数据。

写 WAV 文件的方法和读类似，下面的程序将频率扫描波写入 WAV 文件：



write_wave.py
将频率扫描波写入 WAV 文件

```
import wave
import numpy as np
import scipy.signal as signal

framerate = 44100
time = 10

# 产生 10 秒 44.1 kHz 的 100 Hz-1 kHz 的频率扫描波
t = np.arange(0, time, 1.0/framerate)
wave_data = signal.chirp(t, 100, time, 1000, method='linear') * 10000 ❶
wave_data = wave_data.astype(np.short)

# 打开 WAV 文档
f = wave.open(r"sweep.wav", "wb")

# 配置声道数、量化位数和取样频率
f.setnchannels(1) ❷
f.setsampwidth(2)
f.setframerate(framerate)
# 将 wav_data 转换为二进制数据写入文件
f.writeframes(wave_data.tostring()) ❸
```

```
f.close()
```

❶调用 SciPy 的 signal 模块中的 chrip()，产生长度为 10 秒、取样频率为 44.1 kHz、100 Hz 到 1 kHz 的频率扫描波。由于 chrip()返回的数组为浮点数类型，因此调用数组的 astype()将其转换为短整型数组。

用"wb"模式打开 WAV 文件之后，❷分别调用 setnchannels()、setsampwidth()、setframerate()方法设置 WAV 文件的声道数、量化位数、取样频率。也可以调用 setparams()一次配置所有参数。最后❸调用 writeframes()方法，将数组的二进制数据写入文件。

用 Python 的标准模块读写 WAV 文件时，需要调用 NumPy 的相关函数将字节数据转换为真正的声音数据。SciPy 提供了快速读写 WAV 文件的函数，能直接返回数组。下面的语句读取 Windows XP 的警告声文件，read()返回两个值——取样频率和表示声音数据的数组：

```
>>> import scipy.io.wavfile as wavfile
>>> rate, data = wavfile.read(r"c:\WINDOWS\Media\ding.wav")
>>> rate
22050
>>> data.shape
 (20191, 2)
>>> data.dtype
dtype('int16')
```

由上面的结果可知，“ding.wav”的取样频率为 22050 Hz，而声音数据有两个声道、20191 个取样值，每个取样值都用短整型数表示。下面的语句将声音数据倒转，然后写入“reverse_ding.wav”文件：

```
>>> wavfile.write("reverse_ding.wav", rate, data[::-1, :])
```

虽然用 wavfile 模块能快速读写声音文件，但是它会一次将所有的数据都读入内存，因此不适合处理很大的 WAV 文件。用前面介绍的方法可以分段读取声音信号数据，便于程序处理。

13.1.2 用 pyAudio 播放和录音

使用 pyAudio 模块可以直接利用声卡实时地进行声音数据的输入和输出。

pyAudio 模块对开源声音库 PortAudio 的一些常用功能进行了简单封装，目前它只支持阻塞式的输入输出模式。所谓阻塞模式，是指需要用户程序主动读写输入输出流。虽然阻塞模式在功能上有一定局限，但是由于编程比较简单，因而非常适合编写处理声音的脚本程序。



http://people.csail.mit.edu/hubert/pyaudio
pyAudio 模块的下载地址

下面先看看如何播放声音：



pyaudio_play.py
用 pyAudio 播放 WAV 文件

```
import pyaudio
import wave

chunk = 1024
wf = wave.open(r"c:\WINDOWS\Media\ding.wav", 'rb')
p = pyaudio.PyAudio() ❶
# 打开声音输出流
stream = p.open(format = p.get_format_from_width(wf.getsampwidth()), ❷
                channels = wf.getnchannels(),
                rate = wf.getframerate(),
                output = True)

# 写声音输出流到声卡进行播放
while True:
    data = wf.readframes(chunk) ❸
    if data == "": break
    stream.write(data) ❹

stream.close()
p.terminate()
```

❶首先创建一个 PyAudio 对象，然后❷调用其 open()方法获得声卡的声音输出流，这里根据 WAV 文件的量化格式、声道数和取样频率，分别配置 open()的各个参数。然后❸循环地从 WAV 文件读取数据，❹并调用声音输出流的 write()方法将读取的数据输出到声卡。在循环体中没有任何等待的代码，因为 pyAudio 使用阻塞模式，因此当底层的输出数据缓存区没有空间保存数据时，“stream.write(data)”会阻塞用户程序，直到有足够空间容纳新的数据为止。

下面列出 PyAudio 对象的 open()方法的参数：

- rate：声音流的取样频率。
- channels：声音流的声道数。
- format：取样值的量化格式，值可以为 paFloat32、paInt32、paInt24、paInt16、paInt8 等。在本例中，使用 get_format_from_width()将 wf.getsampwidth()的返回值 2 转换为 paInt16。
- input：输入流标志，True 表示开启输入流。
- output：输出流标志，True 表示开启输出流。
- input_device_index：输入流所使用的设备编号，如果不指定，将使用系统的默认设备。
- output_device_index：输出流所使用的设备编号，如果不指定，将使用系统的默认设备。
- frames_per_buffer：底层缓存区的大小。

- start：是否立即开启输入输出流，默认值为 True。

从声卡读数据和写数据一样简单，下面是一个简单的检测声音的例子：



pyAudio_record.py
从声卡读取声音数据

```
from pyaudio import PyAudio, paInt16
import numpy as np
from datetime import datetime
import wave

# 将 data 中的数据保存到名为 filename 的 WAV 文件中
def save_wave_file(filename, data):
    wf = wave.open(filename, 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(2)
    wf.setframerate(SAMPLING_RATE)
    wf.writeframes("".join(data))
    wf.close()

NUM_SAMPLES = 2000      # pyAudio 内部缓存的块的大小 ❶
SAMPLING_RATE = 8000    # 取样频率
LEVEL = 1500            # 声音保存的阈值
COUNT_NUM = 20          # NUM_SAMPLES 个取样之内出现 COUNT_NUM 个值大于 LEVEL 的取样，就记
                          录声音
SAVE_LENGTH = 8         # 声音记录的最小长度：SAVE_LENGTH * NUM_SAMPLES 个取样

# 开启声音输入
pa = PyAudio()
stream = pa.open(format=paInt16, channels=1, rate=SAMPLING_RATE, input=True,
                 frames_per_buffer=NUM_SAMPLES)

save_count = 0
save_buffer = []

while True:
    # 读入 NUM_SAMPLES 个取样
    string_audio_data = stream.read(NUM_SAMPLES)
    # 将读入的数据转换为数组
    audio_data = np.fromstring(string_audio_data, dtype=np.short)  ❷
    # 计算大于 LEVEL 的取样的个数
    large_sample_count = np.sum( audio_data > LEVEL )
    print np.max(audio_data)
    # 如果个数大于 COUNT_NUM，至少保存 SAVE_LENGTH 个块
```

```
        if large_sample_count > COUNT_NUM:
            save_count = SAVE_LENGTH
        else:
            save_count -= 1

        if save_count < 0:
            save_count = 0

        if save_count > 0:
            # 将要保存的数据存放到 save_buffer 中
            save_buffer.append( string_audio_data )
        else:
            # 将 save_buffer 中的数据写入 WAV 文件，WAV 文件的文件名是保存的时间
            if len(save_buffer) > 0:
                filename = datetime.now().strftime("%Y-%m-%d_%H_%M_%S") + ".wav"
                save_wave_file(filename, save_buffer)
                save_buffer = []
                print filename, "saved"
```

❶程序中用一些全局变量来配置录音参数：以 SAMPLING_RATE 为采样频率，每次读入一块有 NUM_SAMPLES 个采样的数据块，当读入的采样数据中有 COUNT_NUM 个值大于 LEVEL 时，将数据保存到 WAV 文件中，一旦开始保存数据，保存的数据长度最短为 SAVE_LENGTH 个块。WAV 文件以保存时的时间作为文件名。

❷从声卡读入的数据和从 WAV 文件读入的数据相同，都是二进制数据。由于使用 paInt16 格式(16 位的短整型)保存采样值，因此在将它转换为数组时，设置 dtype 参数为 np.short。

13.2　视频的输入输出

13.2.1　读写视频文件

本节介绍如何在 Windows 系统下使用 OpenCV 进行视频文件的读写。虽然在不同的操作系统下，OpenCV 读写视频的内部实现不同，但是它们的 API 函数的用法是相同的。

在 Windows 下 OpenCV 使用“Video For Windows”(VFW)处理视频文件，它是一种过时的技术，目前已经被 DirectShow 取代，但是由于其用法简单，用它编写简单的视频处理脚本程序还是很合适的。下面我们先看看如何生成视频文件。



aviwrite_waterwave.py

制作水波动画的视频文件

```
import numpy as np
import pyopencv as cv

class WaterWave(object): ❶
    def __init__(self, w, h, N, damping):
        self.width, self.height = w, h
        self.N = N
        self.damping = damping
        self.w1 = np.zeros((self.height, self.width), dtype=np.float)
        self.w2 = np.zeros((self.height, self.width), dtype=np.float)
        self.tmpbuf = np.zeros((self.height-2, self.width-2), dtype=np.float)
        self.bmp = np.zeros((self.height, self.width, 3), dtype=np.uint8)

        h, w = self.height - 2, self.width - 2
        self.slice = [(slice(i,h+i), slice(j,w+j))
            for i in xrange(3) for j in xrange(3) if i!=1 or j!=1]

    def step(self):
        y = np.random.randint(1, self.height-1, self.N)
        x = np.random.randint(1, self.width-1, self.N)
        self.w1[y, x] = np.random.random(self.N) * 120 + 128

        self.tmpbuf[:] = 0
        for s in self.slice:
            self.tmpbuf += self.w1[s]

        self.tmpbuf /= 4
        self.w2[1:-1, 1:-1] *= -1
        self.w2[1:-1, 1:-1] += self.tmpbuf
        self.w2 *= self.damping
        self.w1, self.w2 = self.w2, self.w1

        self.bmp[:,:,0] = self.w1 + 128
        self.bmp[:,:,1] = self.bmp[:,:,0] - (self.bmp[:,:,0] >> 2)
        self.bmp[:,:,2] = self.bmp[:,:,1]
        return self.bmp

WIDTH, HEIGHT = 640, 480
video = cv.VideoWriter() ❷
video.open("waterwave.avi", cv.CV_FOURCC(*"DIB "), 30, cv.Size2i(WIDTH, HEIGHT)) ❸
water = WaterWave(WIDTH, HEIGHT, 100, 0.97)
import time
start = time.clock()
for i in xrange(200):
    if i % 30 == 0: print i
    r = water.step()
```

```
        mat = cv.asMat(r)
        video << mat ❹
    del video ❺
    print time.clock() - start
```

❶WaterWave 类用来模拟如图 13-2 所示的水波效果。其 step()方法返回一个表示水波图像的 NumPy 数组。关于水波动画的计算原理请感兴趣的读者搜索关键词“实现水波的模拟”。

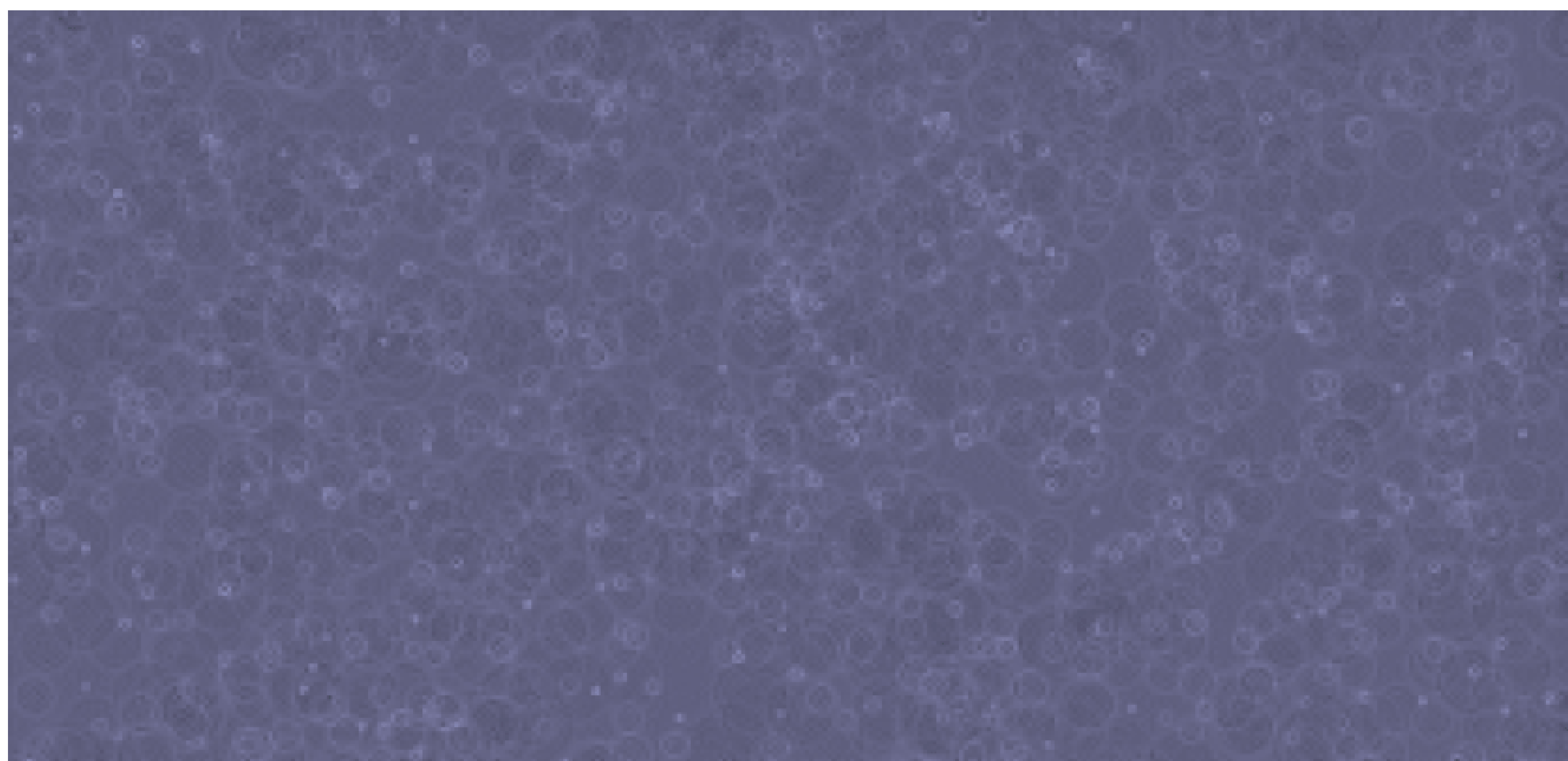


图 13-2　水波动画的截图

❷我们使用 PyOpenCV 的 VideoWriter 类将 WaterWave 对象的计算结果输出到 AVI 视频文件中。❸VideoWriter 对象的 open()方法有 4 个参数：视频文件名、表示视频压缩格式的 FOURCC 参数、每秒的帧数、视频画面的大小。这里的 FOURCC 参数是一个用 CV_FOURCC()计算的表示 4 个字符的 32 位整数，程序中的"DIB "表示所生成的视频不进行任何压缩，因此产生的视频文件会很大。13.3.2 节将对 FOURCC 参数进行详细介绍，读者可以选择安装合适的视频编码库，从而制作更小的视频文件。

❹使用左移位运算符“<<”将表示图像的 Mat 对象添加到视频文件中，❺最后在 VideoWriter 对象销毁时完成视频文件的输出。

由上面的例子可知，只要我们能创建一系列表示图像的数组，就可以用 VideoWriter 对象将这些数组转换为视频文件。由于 matplotlib 提供了访问图表的图像数据的方法，因此不需要将图表保存到临时的图像文件中，而是可以直接将图表的内部图像数据转换成数组，从而快速制作视频文件。下面是一个例子：



aviwrite_matplotlib.py

将 matplotlib 绘制的心形隐函数曲线制作成动画

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import pyopencv as cv

y, x = np.ogrid[-2:2:300j,-2:2:300j]
z = (x**2+y**2-1)**3 - x**2*y**3

fig = plt.figure(figsize=(4,4))
w, h = fig.bbox.width, fig.bbox.height

video = None
for level in np.linspace(-0.2,0.2,101):
    fig.clear() ❶
    axe = fig.add_subplot(111, aspect=1)
    plt.contour(x.ravel(), y.ravel(), z, levels=[level])
    plt.title("level=%5.3f" % level)
    axe.xaxis.set_ticks([])
    axe.yaxis.set_ticks([])
    fig.canvas.draw() ❷
    buf = fig.canvas.buffer_rgba(0,0) ❸
    array = np.frombuffer(buf, np.uint8) ❹
    array.shape = h, w, 4

    if not video:
        video = cv.VideoWriter()
        size = cv.Size2i(int(w),int(h))
        video.open("contour.avi", cv.CV_FOURCC(*"DIB "), 30, size)
        image = cv.Mat(size, cv.CV_8UC3)
    image[:] = array[:,:,2::-1] ❺
    video << image
del video
```

程序中通过循环调用 contour()来绘制一系列等值线。为了重复使用同一个图表对象 fig，❶每次绘图之前用 fig.clear()将图表内容清空，然后再调用各种绘图函数进行绘图。❷绘图函数并不会立即绘图，因此在保存数据之前，需要调用图表的 canvas 属性的 draw()方法强制图表进行绘图。

❸canvas 属性中保存有图表的图像数据，通过其 buffer_rgba()方法可以获得保存图像数据的 buffer 对象，它的两个参数是 buffer 对象在图像数据中的起始点坐标。图像的大小可以通过 canvas 属性的 size()获得，也可以通过图表对象的 bbox 属性获得。❹调用 frombuffer()将 buffer 对象转换为数组，然后对数组的形状进行修改。数组的第 0 轴的长度为图像的高度，第 1 轴的长度为图像的宽度，第 2 轴的长度为图像的通道数。

❺由于 OpenCV 保存的图像的通道按照蓝、绿、红的顺序排列，因此需要对数组的第 2 轴进行反转，并且去除 alpha 通道。

如果只使用 matplotlib 的绘图功能，而不需要显示任何绘图界面，那么可以在程序开头添加下面两行代码，让 matplotlib 使用不显示界面的 Agg 后台绘图库进行绘制。

```
import matplotlib
matplotlib.use('Agg')
```

图 13-3 显示了从所生成的视频文件中截取的 3 帧图像。

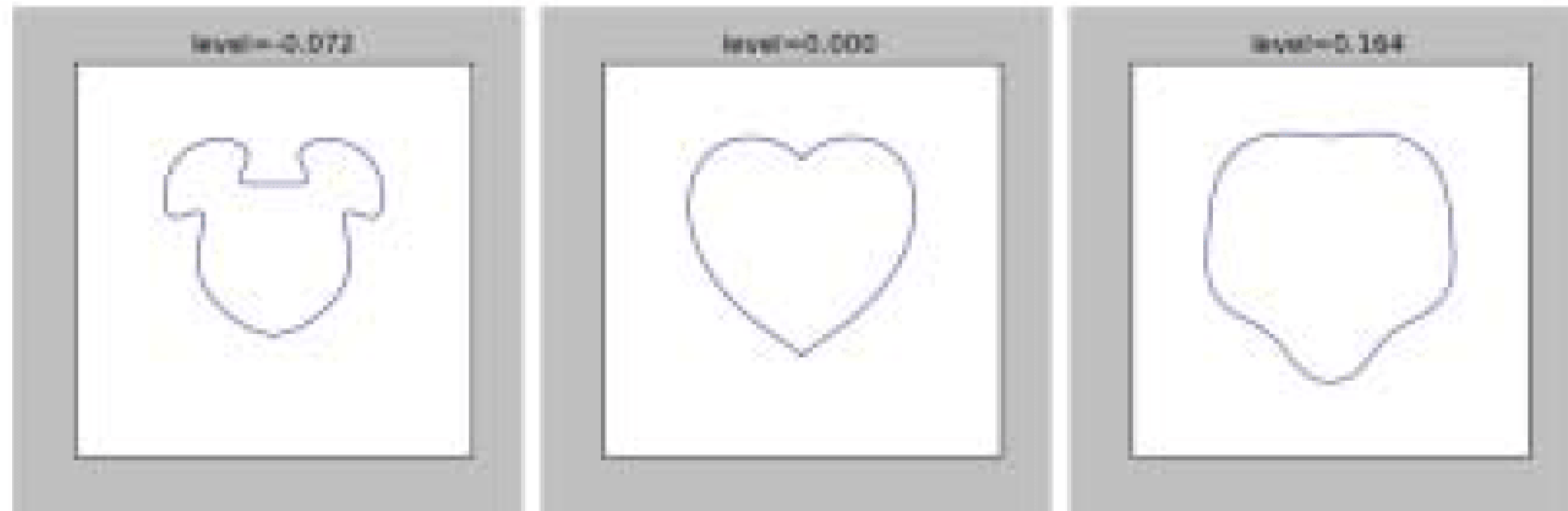
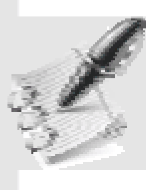


图 13-3　心形隐函数曲线动画中的 3 帧图像

使用 VideoCapture 对象可以从视频文件或视频设备(例如 USB 摄像头)获取图像，下面是播放水波视频文件的程序。



aviread.py
从视频文件或视频设备读取图像数据

```
import pyopencv as cv

def show_video(fileorid):
    cv.namedWindow(str(fileorid), cv.CV_WINDOW_AUTOSIZE)
    video = cv.VideoCapture(fileorid) ❶
    img = cv.Mat() ❷
    while video.grab(): ❸
        video.retrieve(img, 0) ❹
        cv.imshow(str(fileorid), img)
        cv.waitKey(5)

if __name__ == "__main__":
    import sys
    try:
        fileorid = sys.argv[1]
        if fileorid.isalnum():
            fileorid = int(fileorid)
    except:
        fileorid = "waterwave.avi"
    show_video(fileorid)
```

❶首先创建 VideoCapture 对象，它的参数可以是一个表示视频文件名的字符串，也可以是一个表示视频设备的整数。❷创建一个空的 Mat 对象，以保存 VideoCapture 对象获得的图像。❸调用 grab()方法捕捉视频中的下一帧图像，如果返回 False，就表示没有更多的帧了。❹调用 retrieve()方法将捕捉的图像保存到 img 对象中，第二个参数只有在多通道摄像头时才起作用，通常它的值为 0。

VideoCapture 对象还提供了 get()和 set()方法，以获取和设置一些视频相关的属性。例如：

```
>>> import pyopencv as cv
>>> v = cv.VideoCapture("waterwave.avi")
>>> v.get(cv.CV_CAP_PROP_FRAME_COUNT) # 获得视频的总帧数
200.0
>>> v.get(cv.CV_CAP_PROP_FRAME_WIDTH) # 获得视频的宽度
640.0
```


每个视频相关属性都用一个整数表示，在 OpenCV 中它们的预定义名称都是以“CV_CAP_PROP”开头，读者可以使用 IPython 的自动补全功能查看所有的视频属性名。这些属性有些是可以通过 set()方法进行设置的，例如：

```
>>> v.set(cv.CV_CAP_PROP_POS_FRAMES, 100) # 设置视频的当前帧为第 100 帧
```

13.2.2 安装视频编码

虽然 VFW 是过时的技术，但是很多视频编码解码库仍然支持 VFW 接口。如果读者的计算机上安装了这些视频编码解码库，就可以通过设置 FOURCC 参数使用它们进行视频文件的读写。

读者可以通过下面的链接下载 ffdshow 视频编码解码库，使用默认设置即可安装“VFW 接口”。



http://ffdshow-tryout.sourceforge.net
支持 VFW 接口的视频编码解码库

每个视频编码都可以使用特定的 FOURCC 参数进行选择，如果无法确定系统中安装的视频编码，可以用-1 作为 FOURCC 参数传递给 VideoWriter 对象的 open()方法。这样将会打开一个如图 13-4 所示的视频压缩对话框。通过此对话框选择“ffdshow Video Codec”为压缩程序，并单击“配置”按钮打开图中显示的 ffdshow 视频编码器设置对话框，在这里可以对视频编码的一些参数进行配置。视频编码中最重要的参数就是码率，码率越高，视频越清晰，但是视频文件也越大。

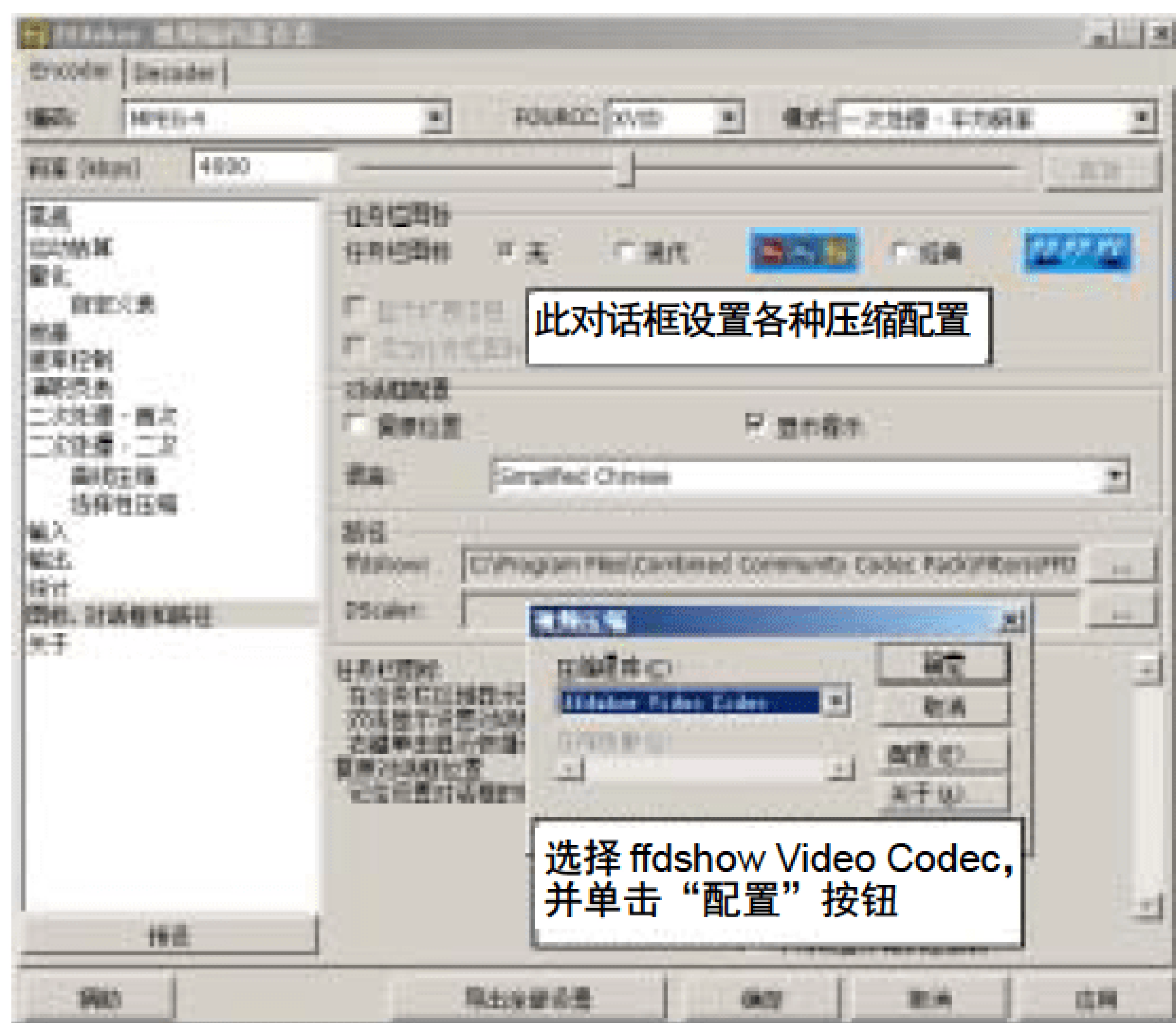


图 13-4　视频压缩对话框和编码器设置对话框

但是，VideoWriter 对象没有提供获取所选视频编码的 FOURCC 参数的方法，因此每次运行时都需要通过对话框来选择编码。为了解决这个问题，本书提供了下面的小工具程序，以帮助读者查看视频编码的 FOURCC 参数。此程序通过 ctypes 库直接调用 VFW 的 DLL(Dynamic Link Library，动态链接库)，显示视频编码对话框，并输出所选中编码的 FOURCC 参数。



avifile.py
打开视频编码对话框，并显示所选编码的 FOURCC 参数

通过上面的程序可知，ffdshow 对应的 FOURCC 参数是"ffds"。ffdshow 的视频编码设置对话框也可以通过 Windows“开始”菜单中的“ffdshow”→“VFW configuration”进行配置。读者可以将 13.2.1 节介绍的制作水波动画程序中的 FOURCC 参数修改为"ffds"，并设置不同的视频编码参数来查看视频压缩效果。

13.3　读写 HDF5 文件

HDF5 是存储科学计算数据的一种文件格式，它支持大于 2GB 的文件，可以把它看做针对科学计算的数据库文件。关于 HDF5 文件格式的更多信息，请参考下面的链接：



http://www.nsmc.cma.gov.cn/FENGYUNCast/docs/HDF5.0_chinese.pdf
中文的 HDF5 使用简介

HDF5 文件就像一个保存数据的文件系统，其中只有两种类型的对象——资料数据(dataset)和目录(group)：

- 资料数据(dataset)：像文件系统中的文件一样用于保存各种数据，例如 NumPy 数组。
- 目录(groups)：类似于文件系统中的文件夹，可以包含其他的目录或资料数据。

HDF5 中用于存取对象的路径和文件系统的路径类似，例如：

```
/MyGroup/Group1/Data
```

第一个“/”表示根目录，“MyGroup”和“Group1”为子目录名，“Data”为数据名。

使用 h5py 模块可以很方便地存取 HDF5 文件。通过它可以像使用字典一样操作 HDF5 的目录，像使用 NumPy 数组一样操作 HDF5 的资料数据。

下面的实例演示了如何使用 h5py 模块。



hdf5_example.py
使用 h5py 创建 HDF5 文件



由于不能直接覆盖已经存在的目录和数据，因此在运行“hdf5_example.py”之前，请确保同一目录下没有“tmp.hdf5”文件。

```
>>> import h5py
>>> f = h5py.File("tmp.hdf5")
```

首先载入 h5py 模块，并创建一个 HDF5 文件对象。和一般的 Python 文件对象一样，我们可以指定打开文件的模式为"r"、"w"或"a"，默认为"a"，表示打开的文件可以进行读写操作。

接下来调用 HDF5 文件对象的 create_group()方法创建几个目录，HDF5 文件对象表示的根目录以及所有子目录对象都有 create_group()方法，通过它可以在任何目录中快速创建子目录。目录对象和字典一样，可以通过 keys()方法查询它所包含的对象名：

```
>>> f.create_group("group1")
<HDF5 group "group1" (0 members)>
>>> f.keys()
['group1']
>>> g2 = f.create_group("group2")
>>> f.keys()
['group1', 'group2']
>>> g2 == f["group2"]
True
>>> g2.create_group("subgroup1")
<HDF5 group "subgroup1" (0 members)>
>>> g2.keys()
['subgroup1']
```

上面的语句创建了如下的目录结构：

```
/group1
/group2/subgroup1
```

可以像添加字典元素一样向目录中添加数据：

```
>>> g2["a"] = np.arange(0,5)
>>> g2["a"]
<HDF5 dataset "a": shape (5,), type "<i4">
>>> g2["a"].value
array([0, 1, 2, 3, 4])
```

HDF5 的数据对象可以直接参与计算：

```
>>> np.sum(g2["a"])
10
>>> g2["a"] + np.arange(1, 6)
array([1, 3, 5, 7, 9])
>>> g2["a"][-1] = 10
>>> g2["a"].value
array([ 0,  1,  2,  3, 10])
```

也可以通过目录对象的 create_dataset()方法创建数据，它的 3 个参数分别为资料名、数组的形状、元素类型：

```
>>> d = g2.create_dataset("b", (4,4), np.float)
>>> d[:,0] = 2.0
>>> d.value
array([[ 2.,  0.,  0.,  0.],
       [ 2.,  0.,  0.,  0.],
       [ 2.,  0.,  0.,  0.],
       [ 2.,  0.,  0.,  0.]])
```

如果通过 data 参数传递数组，那么效果和前面演示的直接赋值一样：

```
>>> g2.create_dataset("c", data=np.arange(0, 1.0, 0.2))
<HDF5 dataset "c": shape (5,), type "<f8">
>>> g2["c"].value
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

和字典操作不一样的是，不能直接覆盖已经存在的数据，必须先使用 del 关键字删除现有的数据，这样能防止因误操作覆盖重要的数据：

```
>>> g2["a"] = np.linspace(0, 1, 5)
ValueError: Name already exists (Symbol table: Object already exists)
```

```
>>> del g2["a"]
>>> g2["a"] = np.linspace(0, 1, 5)
>>> g2["a"].value
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

HDF5 文件中的目录和数据都可以拥有元数据，可以通过 attrs 属性访问这些元数据，元数据的操作也和字典类似：

```
>>> g2.attrs
<Attributes of HDF5 object "group2" (0)>
>>> g2.attrs["title"] = "test data"
>>> g2.attrs["test"] = np.arange(5)
>>> g2.attrs
<Attributes of HDF5 object "group2" (2)>
>>> g2.attrs["test"]
array([0, 1, 2, 3, 4])
>>> f.close()
```


如果读者完全安装了 Python(x,y)，那么可以使用 ViTables 工具查看 HDF5 文件的内容，请在“开始”菜单的 Python(x,y)下查找 ViTables 的快捷方式。图 13-5 是使用 ViTables 查看上面所创建的“tmp.hdf5”文件的界面截图，从中可以看到我们创建的 3 个目录——group1、group2、subgroup1，3 个数据——a、b、c，以及 group2 的元数据。

http://vitables.berlios.de/index.html
ViTables 的下载地址

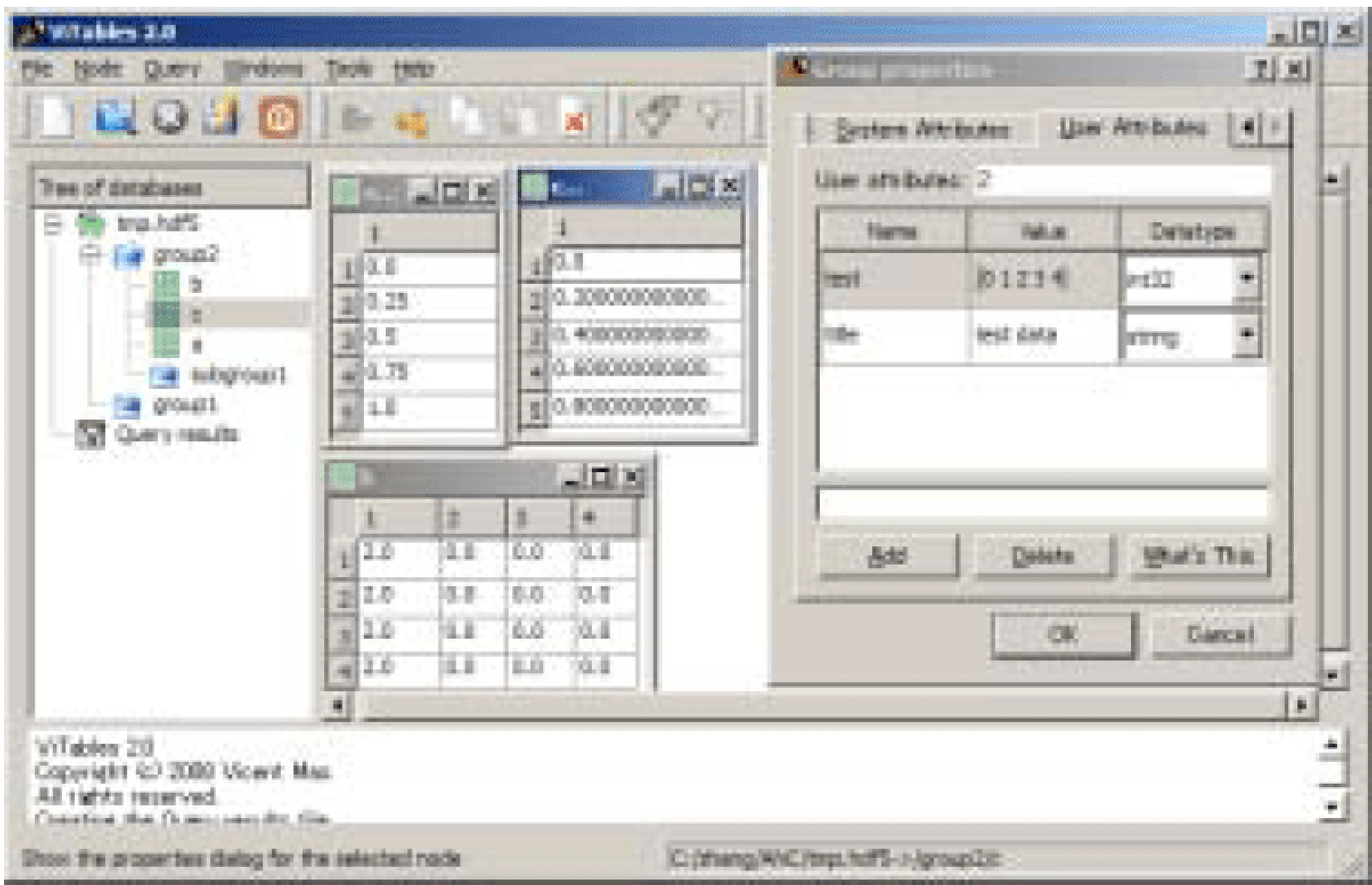


图 13-5　用 ViTables 查看 HDF5 文件的内容

13.4　读写 Excel 文件

本节介绍如何使用 xlrd 和 xlwt 库读写 Excel 文件，使用这些库的好处在于它不需要使用 COM 接口驱动 Microsoft Excel，因此不需要安装微软的 Office 套件就可以操作 Excel 文件。

- xlrd 库的下载地址：http://pypi.python.org/pypi/xlrd。
- xlwt 库的下载地址：http://pypi.python.org/pypi/xlwt。
- xlutils 库的下载地址：http://pypi.python.org/pypi/xlutils。英文帮助文档：http://www.python-excel.org。

13.4.1　写 Excel 文件

先看看如何使用 xlwt 库写 Excel 文件。在 xlwt 中，用 xlwt.Workbook 对象表示 Excel 文件。通过操作此对象可以修改 Excel 文档的内容，最后调用其 save()方法将文档保存成文件。通过操作 Workbook 对象可以创建如下对象：

- WorkSheet：表示 Excel 文档中的一个工作表。可通过 Workbook.add_sheet()创建 WorkSheet 对象，通过 get_sheet()获取已经存在的 WorkSheet 对象。
- Row：表示工作表中的一行，可使用 Worksheet.row()创建或获取。
- Column：表示工作表中的一列，可使用 Worksheet.col()创建或获取。
- Cell：通过 Worksheet.write()或 Row.write()可以直接写指定的单元格。

下面是一个简单的例子：



write_xls.py
演示写 Excel 文件

```
import numpy as np
from xlwt import *

book = Workbook()
sheet1 = book.add_sheet(u'随机数')
head = ["normal", "power", "gamma", "SUM"]
N = 100
data = np.vstack([
    np.random.normal(size=N),
    np.random.power(a=1.0,size=N),
    np.random.gamma(0.9,size=N)
])
# 创建对齐配置
al = Alignment()
al.horz = Alignment.HORZ_CENTER
al.vert = Alignment.VERT_CENTER
# 创建边框配置
```

```
borders = Borders()
borders.bottom = Borders.THICK
# 创建样式
style = XFStyle()
style.alignment = al
style.borders = borders
# 获得第 0 行
row0 = sheet1.row(0)
# 将标题写入第 0 行，使用所创建的样式
for i, text in enumerate(head):
    row0.write(i, text, style=style)  ❶
# 写入随机数
for i, line in enumerate(data):
    for j, value in enumerate(line):
        sheet1.write(j+1, i, value)
# 写求和公式，注意公式中的单元格下标从 1 开始计数
for i in xrange(N):
    sheet1.row(i+1).set_cell_formula( ❷
        3, Formula("sum(A%s:C%s)" % (i+2, i+2)), calc_flags=1)

# 设置 4 个列的宽度
for i in xrange(4):
    sheet1.col(i).width = 4000 ❸
# 设置第 0 行的高度
sheet1.row(0).height_mismatch = 1
sheet1.row(0).height = 1000
book.save("tmp.xls")
```

程序的输出文件如图 13-6 所示。程序中除了将数组写入 Excel 文件之外，还演示了各种属性的配置。

❶通过 write()的 style 参数指定单元格的样式。样式的创建比较麻烦，但也很直观：先分别创建 Aligment、Borders 等对象，配置样式的各个属性，然后创建 XFStyle 对象引用各种配置。❷通过 Formula 对象可以将 Excel 的公式写入文档。❸通过设置 Row 和 Column 对象的 width 和 height 属性可以设置行和列的宽度和高度。

	A	B	C	D
1	normal	power	gamma	SUM
2	-0.091999221	0.270696493	1.483073669	1.661670941
3	-1.051917952	0.051247001	1.236888935	0.235217984
4	0.657499188	0.858595076	0.150992634	1.667086897
5	1.91262384	0.266440983	0.018128269	2.197193093
6	0.366034327	0.188331234	0.880789002	1.435154563
7	0.026145624	0.97312959	2.666193711	3.665469126
8	1.345948204	0.536077419	5.982567519	7.864593141
9	-0.04038639	0.25273177	0.612746112	0.825091491

图 13-6　用 xlwt 输出的 Excel 文件

xlwt 的 Utils 模块中有如下经常会用到的辅助函数：

- col_by_name()：将列名转换为列的编号，例如将"C"转换为 2。
- cell_to_rowcol()：将单元格的字符串表达式变换为行列编号。它返回一个包含 4 个元素的元组——(行编号，列编号，行是否为绝对坐标，列是否为绝对坐标)。
- cell_to_rowcol2()：和 cell_to_rowcol()类似，但是只返回编号信息。
- rowcol_to_cell()：和 cell_to_rowcol()相反，将编号变换为字符串表达式。
- cellrange_to_rowcol_pair()：将字符串表示的范围变换为一个包含 4 个元素的元组——(起始行号，起始列号，结束行号，结束列号)
- rowcol_pair_to_cellrange()：和 cellrange_to_rowcol_pair()相反。

下面是几个例子：

```
>>> from xlwt import Utils
>>> Utils.cell_to_rowcol("C4")
(3, 2, False, False)
>>> Utils.cell_to_rowcol("C8")
(7, 2, False, False)
>>> Utils.cell_to_rowcol2("C8")
(7, 2)
>>> Utils.cell_to_rowcol("$C$8")
(7, 2, True, True)
>>> Utils.rowcol_to_cell(200,100)
'CW201'
>>> Utils.rowcol_to_cell(200,100, row_abs=True, col_abs=True)
'$CW$201'
```

13.4.2 读 Excel 文件

读 Excel 文件需要使用 xlrd 库，下面的程序读入 13.4.1 节输出的“tmp.xls”文件。



xlrd 无法直接读取 xlwd 输出的公式单元格，需要先用 Excel 打开文档并保存之后才能用 xlrd 计算公式单元格。

首先从 xlrd 库导入打开 WorkBook 的函数 open_workbook()，并用它打开“tmp.xls”文件：



read_xls.py
用 xlrd 库读取 Excel 文件

```
>>> from xlrd import open_workbook
>>> book = open_workbook("tmp.xls")
```

接下来可以调用 book 的属性和方法获得我们所需要的信息：

```
>>> book.nsheets # 工作表的数目
1
>>> print book.sheet_names()[0] # 第一个工作表的名字
随机数
>>> sheet = book.sheets()[0] # 获得第一个工作表
```

调用工作表对象的 cell()、row()和 col()，可以读取其中指定的单元格、行以及列中的元素，这些方法返回的是 Cell 对象。如果希望直接获得它们的值，可以调用 cell_value()、row_values()和 col_values()方法：

```
>>> sheet.cell(0, 0) # 读取 A1 的内容
text:u'normal'
>>> sheet.row(0) # 读取第一行的内容
[text:u'normal', text:u'power', text:u'gamma', text:u'SUM']
# 读取第二列的内容，从第二行开始，并对其求和
>>> sum(x.value for x in sheet.col(1, start_rowx=1))
50.080880120307619
>>> sum(sheet.col_values(1, start_rowx=1)) # 同上
50.080880120307619
>>> sheet.cell(1,3) # 读取公式单元格的值
number:0.41405871399469785
```

如果我们需要读取某个 Excel 文件，修改其中的某些内容，然后再写回 Excel 文件中，那么可以使用 xlutils 库的 copy()来快速完成从 xlrd.Book 到 xlwt.WorkBook 的转换复制工作。下面的程序在“tmp.xls”中添加一些文字，然后将结果保存到“output.xls”文件中：



modify_xls.py
使用 xlutils.copy()实现 Excel 文件的修改

```
from xlrd import open_workbook
from xlutils.copy import copy
rb = open_workbook('tmp.xls',formatting_info=True)
wb = copy(rb)
ws = wb.get_sheet(0)
ws.write(0, 4, u"我添加了一些内容")
wb.save('output.xls')
```



xlrd 库无法获取公式本身的字符串，因此公式单元格无法在复制过程中保留下来。

第 14 章

数字信号系统

本章以 FIR 和 IIR 滤波器为核心介绍数字信号系统中的一些基础知识。在介绍其原理的同时，还将用 NumPy 和 SciPy 中的相关函数直观地对原理进行实例演示，以帮助读者理解数字信号处理的一些理论知识。

14.1 FIR 和 IIR 滤波器

在数字信号处理领域中，数字滤波器占有非常重要的地位。根据其计算方式可以分为 FIR(有限脉冲响应)滤波器和 IIR(无限脉冲响应)滤波器两种。

FIR 滤波器的计算公式如下：

$$y[m] = b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[m-P]$$

IIR 滤波器的直接 1 型计算公式如下：

$$\begin{aligned} y[m] = & b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[m-P] \\ & - a[1]y[m-1] - a[2]y[m-2] - \cdots - a[Q]y[m-Q] \end{aligned}$$

其中，x 是输入信号，数组 a 和 b 是滤波器的系数，y 是滤波器的输出。可以把 FIR 滤波器看作 IIR 滤波器的一种特殊情况：当系数 a 都为 0 时，IIR 滤波器就变为 FIR 滤波器。

根据 FIR 滤波器的计算公式可以得知，时刻 m 的输出 y[m]由时刻 m 的输入 x[m]以及之前的输入 x[m-1] ... x[m-P]和滤波器的系数 b[0] ... b[P]求乘积和而得。而 IIR 滤波器只不过是再减去之前的输出 y[m-1] ... y[m-Q]和系数 a[1] ... a[m-Q]的乘积和。

总之，数字滤波器的计算方法并不复杂，仅仅是数组对应元素的乘积与求和而已。然而其计算量对于 Python 来说是相当大的：通常 FIR 滤波器的系数都很长，例如 CD 音质的数字声音信号一秒钟有 44100 个取样值，假设滤波器的长度是 100，那么一秒钟需要计算 4 百万次以上的乘法和加法运算，这对于 Python 这样的动态语言来说是很困难的。

SciPy 的 signal 库提供了 lfilter()来完成数字滤波器的计算工作。由于它的计算是在 C 语言级别实现，因此运算速度很快：

```
signal.lfilter(b, a, x, axis=-1, zi=None)
```

其中，b 和 a 是滤波器的系数，x 是表示输入信号的数组。lfilter()并不直接使用前面介绍的 IIR 滤波器计算公式进行计算，而是对其进行了如下变形：

```
y[m] = b[0]*x[m] + z[0,m-1]                        (1)
z[0,m] = b[1]*x[m] + z[1,m-1] - a[1]*y[m]          (2)
...
z[n-3,m] = b[n-2]*x[m] + z[n-2,m-1] - a[n-2]*y[m]
z[n-2,m] = b[n-1]*x[m] - a[n-1]*y[m]
```

这段公式就没有 IIR 滤波器的直接 1 型公式那么直白了，但是只需要仔细观察一下就不难发现，将式(2)的时间变为 m-1，得到：

```
z[0,m-1] = b[1]*x[m-1] + z[1,m-2] - a[1]*y[m-1]     (3)
```

将式(3)带入式(1)中，发现 b[0]*x[m]、b[1]*x[m-1]、-a[1]*y[m-1]等项是直接 1 型公式中的项。将后续的公式依次代入，最后得到的公式将和直接 1 型公式完全一致，这个计算公式被称为直接 2 型。

在直接 1 型公式中，为了计算 m 时刻的输出 y[m]，除了需要 m 时刻的输入 x[m]之外，还需要 x[m-1] ... x[m-P]以及 y[m-1] ... y[m-Q]，这些值都需要作为滤波器的内部状态保存起来，因此需要保存 P+Q 个数值。而根据直接 2 型公式，只需要保存 z[0] ... z[n-2]即可，其中 n 为系数 a 和 b 的长度的最大值，即 max(P, Q)。

通过 lfilter()的 zi 参数可以指定滤波器的初始状态。当 zi 不为 None 时，lfilter()将返回滤波器的最终状态 zf，于是返回值为(y, zf)。如果 zi 为 None，将只返回滤波器的输出 y。

当使用 lfilter()对很长的输入信号进行滤波计算时，经常需要对数据进行分段处理。这时就需要将 lfilter()返回的表示滤波器状态的数组 zf，作为参数传递给下一次函数调用。下面的程序演示了这种分段滤波的方法。



filter_lfilter_example.py
用 lfilter 进行分段滤波

```
# 某个均衡滤波器的参数
a = np.array([1.0, -1.947463016918843, 0.9555873701383931])
b = np.array([0.9833716591860479, -1.947463016918843, 0.9722157109523452])

# 44.1k Hz，1 秒的频率扫描波
t = np.arange(0, 0.5, 1/44100.0)
x= signal.chirp(t, f0=10, t1 = 0.5, f1=1000.0)

# 直接计算滤波器的输出
y = signal.lfilter(b, a, x)

# 将输入信号分为 50 个数据一组
```

```
x2 = x.reshape(-1,50)

# 滤波器的初始状态为0，长度是滤波器系数的长度-1
z = np.zeros(max(len(a),len(b))-1)
y2 = [] # 保存输出的列表

for tx in x2:
    # 对每段信号进行滤波，并更新滤波器的状态 z
    ty, z = signal.lfilter(b, a, tx, zi=z)
    # 将输出添加到输出列表中
    y2.append(ty)

# 将输出 y2 中的多个数组链接成一个一维数组
y2 = np.hstack(y2)

# 输出 y 和 y2 之间的误差
print np.sum((y-y2)**2)
```

程序输出的误差为 0，经过均衡滤波器之后的频率扫描波形如图 14-1 所示。

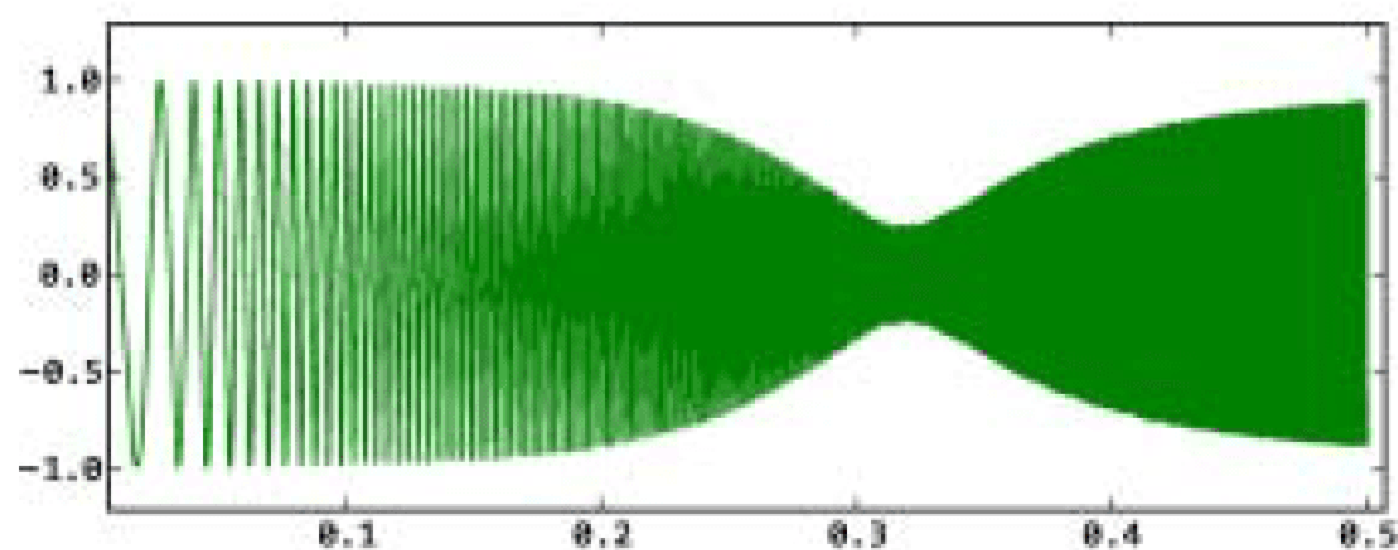


图 14-1　经过均衡滤波器之后的频率扫描波形

程序中使用的 IIR 滤波器为二次均衡滤波器，其系数的设计算法将在后面的章节介绍。为了观察滤波器的频率特性，我们让它对频率扫描波进行处理。程序中同时采用单次滤波和分段滤波两种方式调用 lfilter()，可以看到它们的结果完全一样。使用分段滤波结合 pyAudio 库，我们很容易写出对实时声音信号进行滤波的程序。

如果将一个脉冲信号输入到滤波器中，那么得到的输出被称为滤波器的脉冲响应。所谓脉冲信号，就是在时刻 0 为 1、在其余时刻均为 0 的信号。根据 FIR 滤波器的公式，FIR 滤波器的脉冲响应就是滤波器的系数。而 IIR 滤波器的脉冲响应就不是很直观了，下面使用 lfilter()计算二次均衡滤波器的脉冲响应，结果如图 14-2 所示。



filter_lfilter_impulse01.py
计算 IIR 均衡滤波器的脉冲响应

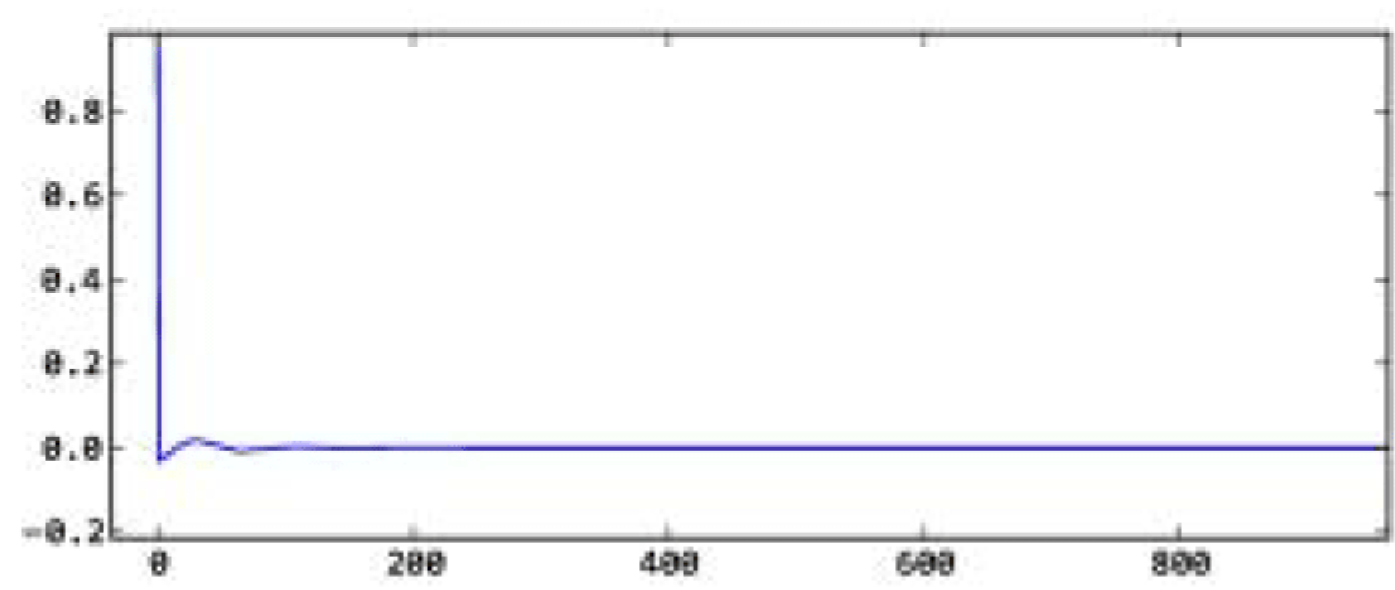


图 14-2　均衡滤波器的脉冲响应


```
>>> run filter_lfilter_example.py # 运行前面的例子，得到滤波器系数 a 和 b
>>> impulse = np.zeros(1000)
>>> impulse[0] = 1
>>> h = signal.lfilter(b, a, impulse)
>>> h[-1]
-4.2666825205952273e-12
```

如果观察 h 的具体数值就会发现，随着时间的推移，h 越来越小，但是始终不会为 0，其脉冲响应是无限长度的，因此才被称为无限脉冲响应滤波器。如果将 h 当做 FIR 滤波器的系数对信号 x 进行滤波，得到的结果和 IIR 滤波器的结果将十分接近：

```
>>> y3 = signal.lfilter(h, 1, x)
>>> np.sum((y-y3)**2)
3.7835244127856444e-17
```

显然，由于 h 是逐渐衰减的，只要测量足够长的脉冲响应，就可以用 FIR 滤波器足够精确地模拟 IIR 滤波器。图 14-3 显示的是误差和 FIR 滤波器的长度之间的关系。由于 IIR 滤波器的脉冲响应呈指数衰减，因此精度随着长度呈指数增加，请注意 Y 轴是对数坐标。

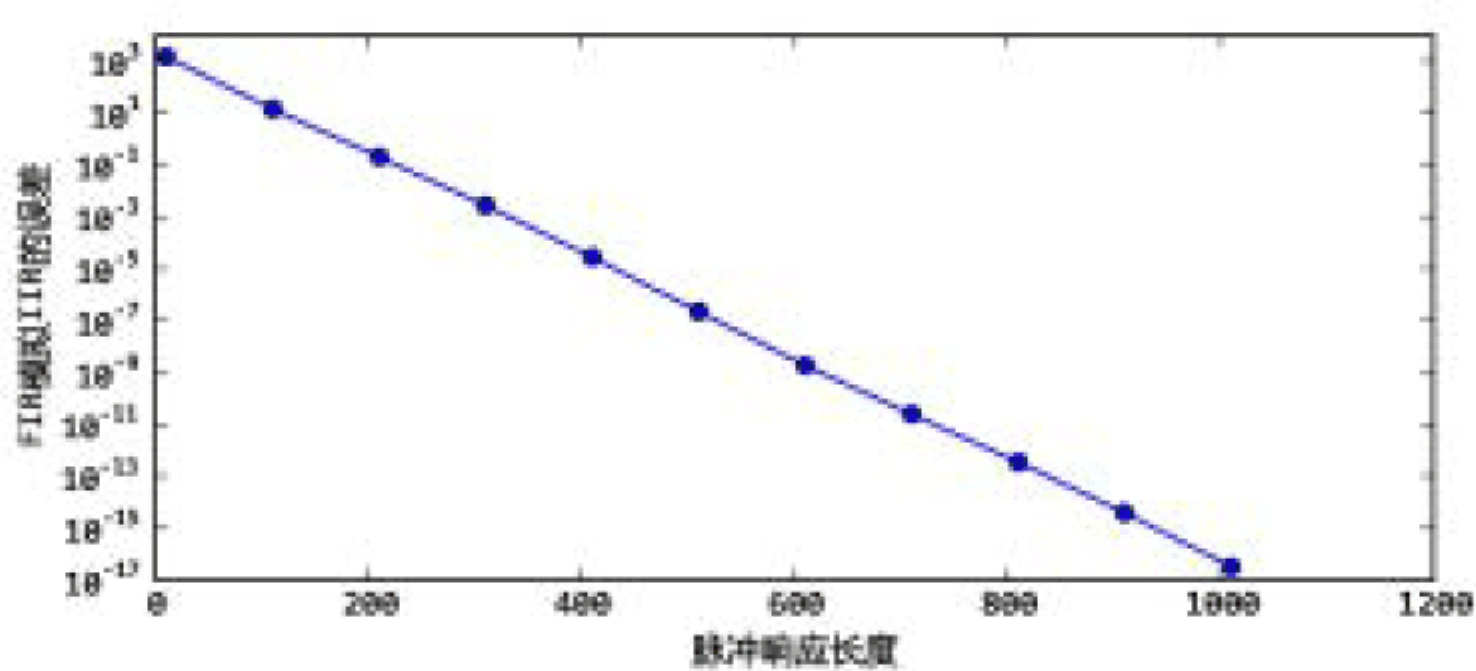


图 14-3　随着 FIR 滤波器的长度的增加，误差呈指数减小

图 14-3 的计算程序如下，用 lfilter()计算 FIR 滤波器的输出时，需要将参数 a 设置为 1：



filter_lfilter_impulse02.py
用 FIR 滤波器模拟 IIR 滤波器

```
import scipy.signal as signal
import numpy as np
import pylab as pl

# 某个均衡滤波器的参数
a = np.array([1.0, -1.947463016918843, 0.9555873701383931])
b = np.array([0.9833716591860479, -1.947463016918843, 0.9722157109523452])

# 44.1k Hz, 1 秒的频率扫描波
t = np.arange(0, 0.5, 1/44100.0)
x= signal.chirp(t, f0=10, t1 = 0.5, f1=1000.0)
y = signal.lfilter(b, a, x)
ns = range(10, 1100, 100)
err = []

for n in ns:
    # 计算脉冲响应
    impulse = np.zeros(n, dtype=np.float)
    impulse[0] = 1
    h = signal.lfilter(b, a, impulse)

    # 直接 FIR 滤波器的输出
    y2 = signal.lfilter(h, 1, x)

    # 输出 y 和 y2 之间的误差
    err.append(np.sum((y-y2)**2))

# 绘图
pl.figure(figsize=(8,4))
pl.semilogy(ns , err, "-o")
pl.xlabel(u"脉冲响应长度")
pl.ylabel(u"FIR 模拟 IIR 的误差")
pl.show()
```

14.2 FIR 滤波器设计

理想的低通滤波器的频率响应如图 14-4 所示。其中，f_s 为取样频率，f_c 为阻带频率。通常

为了计算方便，将取样频率正规化为 1。于是 f_c 的含义就是每个取样点所包含信号的周期数，例如 0.1 表示每个取样点包含 0.1 个周期，即一个周期有 10 个取样点。根据傅立叶变换公式，可以求出此理想低通滤波器的脉冲响应为：

$$h_{ideal}(n) = \frac{\sin(2\pi f_c)}{\pi n} = 2f_c sinc(2f_c n)$$

其中，n 为负无穷到正无穷的整数。显然，此脉冲响应不但无限长，而且不满足因果律，因为对于在 0 时刻输入的脉冲信号，输出信号在 0 时刻之前就有响应了。

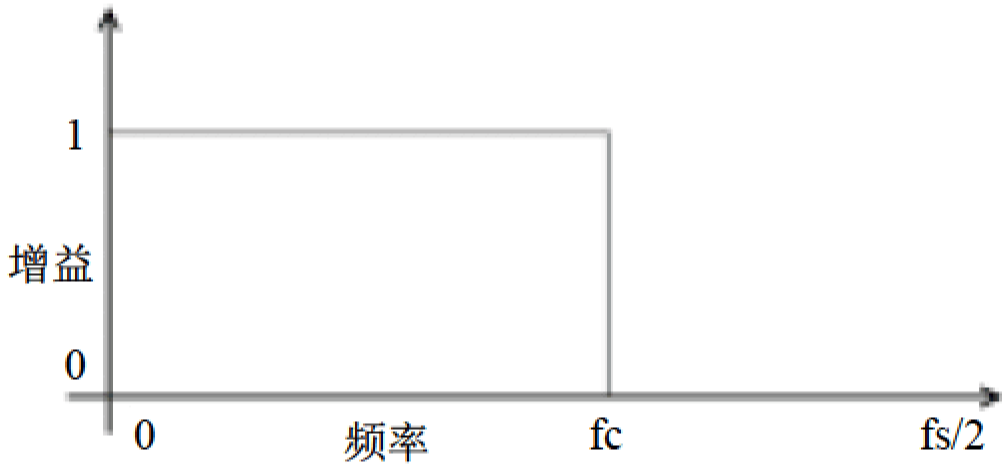


图 14-4　理想低通滤波器的频率响应

这样的脉冲响应当然无法用 FIR 滤波器实现，一个最直观的近似方法就是取 h_{ideal} 中 $0 \leqslant n < L$ 的 L 个值当做低通 FIR 滤波器的系数。下面的程序计算此低通滤波器的频率响应，结果如图 14-5 所示。程序中使用 freqz()计算滤波器的频率响应，并用 20log(| h |)计算以分贝表示的幅值。

filter_firdesign_sinc1.py
用 sinc 函数设计低通 FIR 滤波器，取大于 0 的区间

```
import scipy.signal as signal
import numpy as np
import pylab as pl

def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(0, n, 1.0))

b = h_ideal(30, 0.25)
w, h = signal.freqz(b, 1)
pl.figure(figsize=(8,4))
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)))
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.show()
```

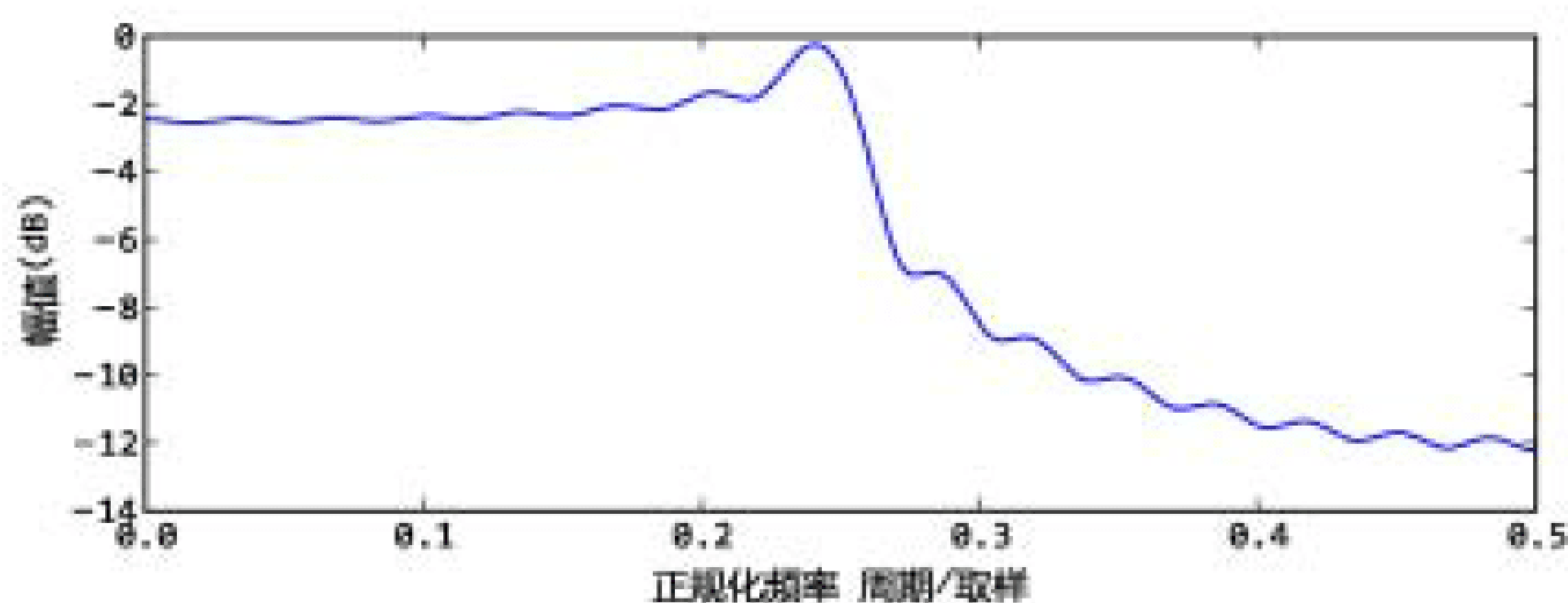


图 14-5 截取 sinc 函数的正时间部分作为脉冲响应的低通滤波器

用 freqz()计算频率响应

freqz()用于计算数字滤波器的频率响应，它的调用方式如下：

```
freqz(b, a=1, worN=None, whole=0, plot=None)
```

其中，b 和 a 是滤波器的系数，worN 为所计算的频率点数，whole 为 0 表示计算频率的上限为 π，whole 为 1 表示计算频率的上限为 2π。

它返回一个元组(w,h)，其中 w 为表示频率的数组，它的值是正规化的圆频率。通过 w/(2*pi)可以计算对应的正规化频率。h 是一个复数数组，其长度和 w 相同，它表示 w 中每个频率对应的频率响应。复数的幅值表示滤波器的增益特性，相角表示滤波器的相位特性。

14.2.1 用 firwin()设计滤波器

显然，前面介绍的低通滤波器的频率响应和理想低通滤波器相差甚远，并且即使增加 FIR 滤波器的系数也没有作用。因为我们舍弃了 n<0 的那一半系数，而这些系数有着相当大的影响。因此只截取 n≥0 的部分是不够的，如果将 n<0 的那一半系数也添加进滤波器系数，那么得到的频率响应将会有很大的改善。按如下所示重新定义 h_ideal()函数，它返回 h_{ideal} 中-n 到 n 之间的系数(图 14-6 是添加 n<0 系数之后的频率响应)：



filter_firdesign_sinc2.py

用 sinc 函数设计低通 FIR 滤波器，取对称区间

```
def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(-n, n, 1.0))
```

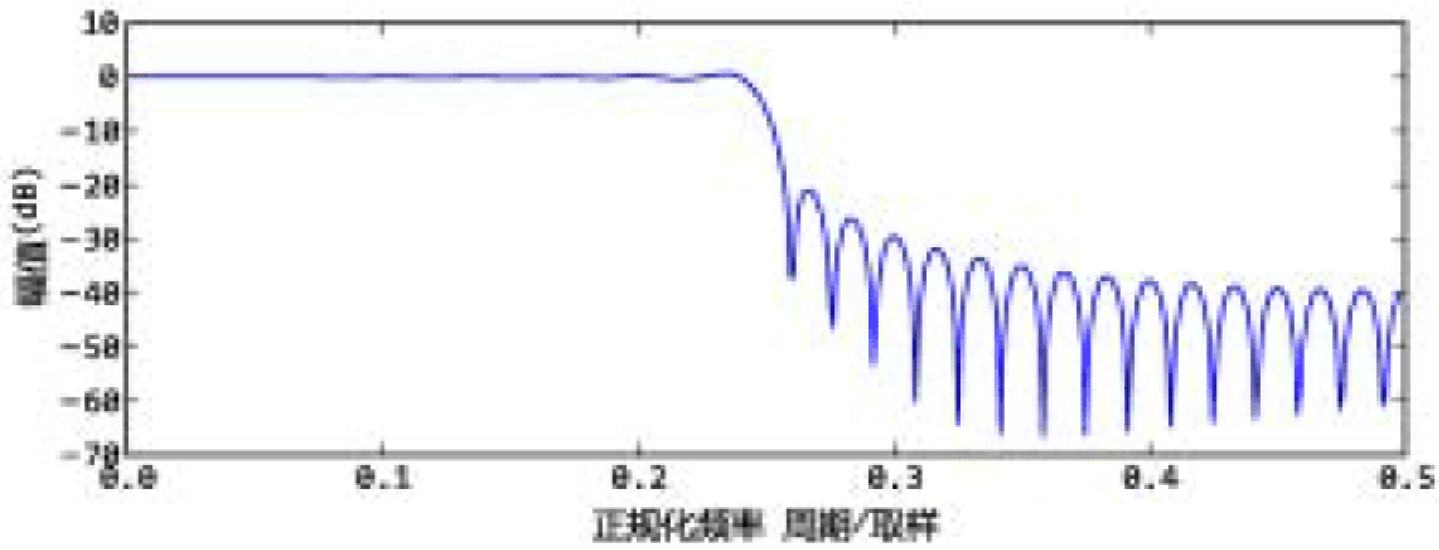


图 14-6　对称截取 sinc 函数的低通滤波器

这样做虽然改善了频率响应，但却给系统带来了许多延时。为了使频率响应更接近理想低通滤波器，必须增加滤波器的点数，然而为了减少延时，又必须减少点数。在设计滤波器时，需要在精度和延时之间进行取舍。为了在同样延时的情况下获得更好的低通滤波效果，可以对截取的滤波器系数进行窗函数处理，让系数的两边能更快地收敛为 0。signal 库提供的 firwin()采用此算法，用窗函数设计低通滤波器，它的调用形式如下：

```
firwin(N, cutoff, width=None, window='hamming')
```

其中，N 为滤波器的长度，cutoff 为以 $f_s/2$ 正规化的频率，window 为使用的窗函数名。

下面的程序用 firwin()设计低通滤波器，并且和上面的结果进行比较。注意，由于 firwin()的 cutoff 参数是以“取样频率/2”正规化的，因此它是前面介绍的 f_c 的两倍，结果如图 14-7 所示(见文前彩插)。

filter_firdesign_firwin.py
使用 firwin()设计 FIR 滤波器

```
import scipy.signal as signal
import numpy as np
import pylab as pl

def h_ideal(n, fc):
    return 2*fc*np.sinc(2*fc*np.arange(-n, n, 1.0))

b = h_ideal(30, 0.25) # 以 fs 正规化的频率
b2 = signal.firwin(len(b), 0.5) # 以 fs/2 正规化的频率

w, h = signal.freqz(b)
w2, h2 = signal.freqz(b2)

pl.figure(figsize=(8,6))
pl.subplot(211)
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)), label=u"h_ideal")
```

```
    pl.plot(w2/2/np.pi, 20*np.log10(np.abs(h2)), label=u"firwin")
    pl.xlabel(u"正规化频率 周期/取样")
    pl.ylabel(u"幅值(dB)")
    pl.legend()
    pl.subplot(212)
    pl.plot(b, label=u"h_ideal")
    pl.plot(b2, label=u"firwin")
    pl.legend()
    pl.show()
```

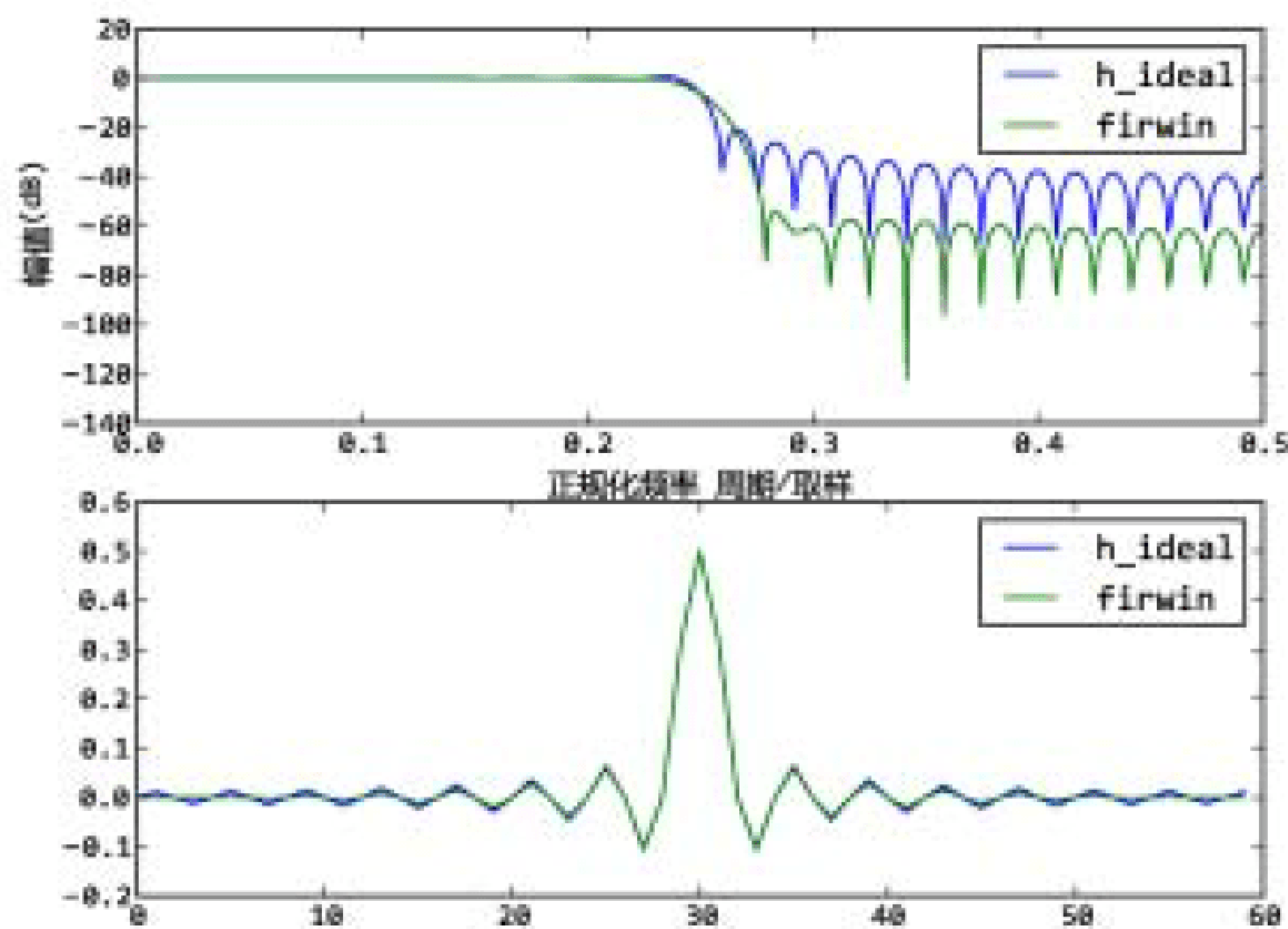


图 14-7　使用 firwin()设计的低通滤波器的频率响应和脉冲响应

使用 firwin()设计的滤波器并不是最优的，为了实现同样的频率响应，还可以使用长度更短的 FIR 滤波器。

14.2.2　用 remez()设计滤波器

remez()能够帮助我们找到更优的滤波器系数，它的调用形式如下：

```
    remez(numtaps, bands, desired,
        weight=None, Hz=1, type='bandpass', maxiter=25, grid_density=16)
```

其中：

- numtaps：所设计的 FIR 滤波器的长度。

- bands：一个递增序列，包括频率响应中所有频带的边界，值在 0 到 Hz/2 之间。如果 Hz 参数为默认值 1，那么可以把它当做以取样频率正规化的频率。
- desired：长度为 bands 参数一半的增益序列，给出 bands 中每个频带的增益值。
- weight：长度和 desired 参数相同的权重序列，给出 desired 中每个增益所占的权重，值越大表示越重要。
- type：可以是'bandpass'或'differentiator'，本书只介绍 type 参数为'bandpass'的情况。

remez 算法

remez 是一种迭代算法，它能够找到一个 n 阶多项式，使得在指定区间中，此多项式和指定函数之间的最大误差最小化。由于 FIR 滤波器的频率响应实际上是一个多项式函数，因此可以用 remez 算法进行 FIR 滤波器系数的设计。

remez()返回最满足指定条件的 FIR 滤波器系数，和 firwin()一样，滤波器的系数总是对称的。当 numtaps 参数为偶数时，所设计的滤波器在“取样频率/2”处的响应为 0，因此无法设计出长度为偶数的高通滤波器。

下面的程序绘制由 remez()设计的高通滤波器的频率响应：



filter_remez.py
使用 remez()设计 FIR 高通滤波器

```
import scipy.signal as signal
import numpy as np
import pylab as pl

for length in [11, 31, 51, 101, 201]:
    b = signal.remez(length, (0, 0.18,  0.2,  0.50), (0.01, 1))
    w, h = signal.freqz(b, 1)
    pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)), label=str(length))
pl.legend()
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.show()
```

程序中，remez()的 bands 参数设置两组以取样频率正规化的频带：0 到 0.18、0.2 到 0.5。desired 参数设置两个频带的增益分别为 0.01 和 1。因此这里设计的是一个通带频率为 0.2、阻带增益为-40dB 的高通滤波器。

程序的运行结果如图 14-8 所示，可以看出滤波器越长，频率响应越接近设计值(见文前彩插)。

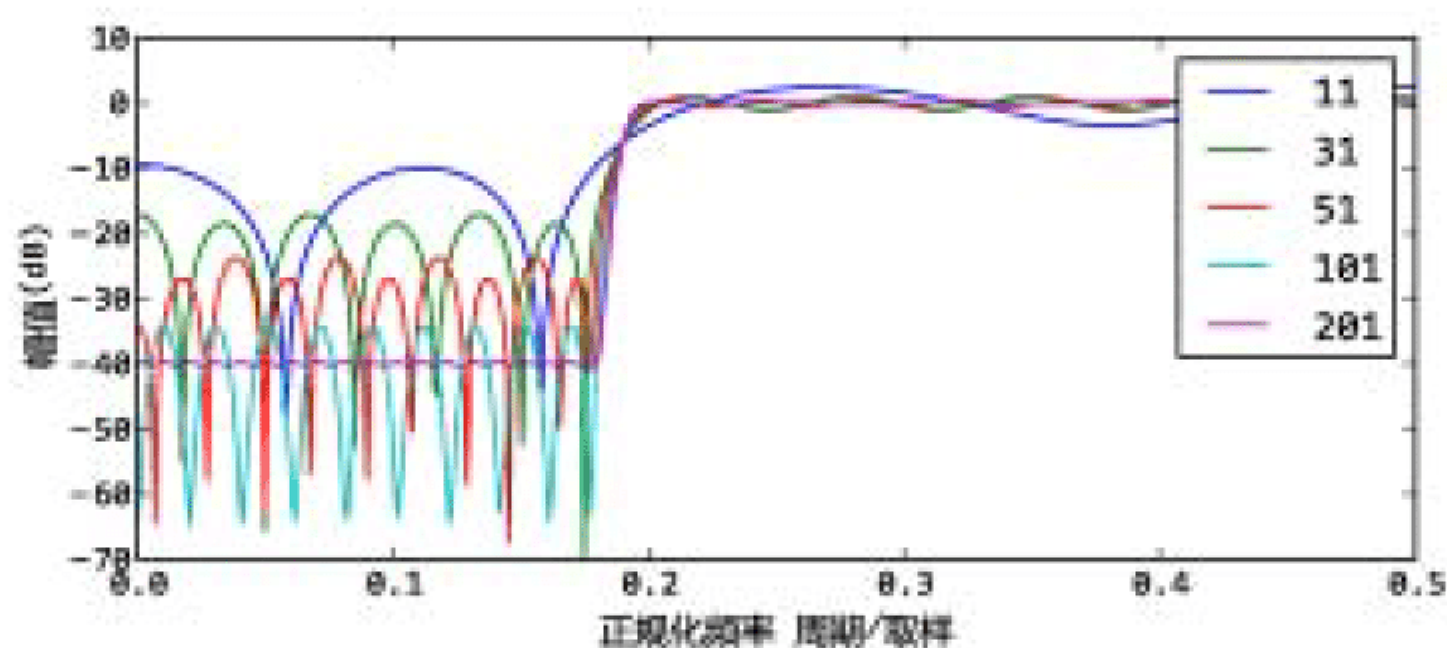


图 14-8　使用 remez()设计的高通滤波器，长度越长，频率响应越接近设计值

图 14-9 显示了权值和频率响应之间的关系(见文前彩插)，图中滤波器的长度为 101。由图可知，当阻带和通带的权值为 1 和 0.01(红色曲线)时，两个频带的增益浮动范围相同，这个权值正好和 desired 参数的设置相反。这是因为在默认情况下，增益越大的频带的频率响应要求越精确，而当权值和增益的乘积相等时，频率响应的误差也就相同了。


filter_remez_weight.py
remez()的权值参数

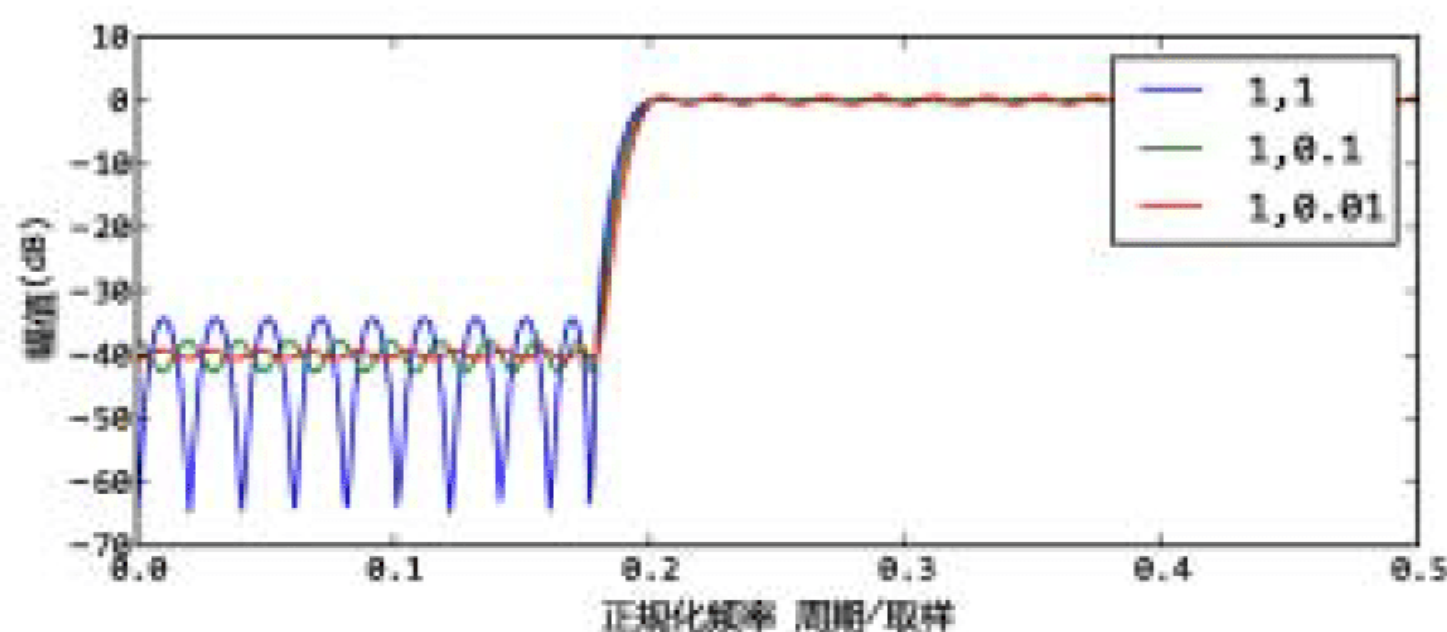


图 14-9　使用 remez()设计滤波器时，权值会影响频率响应

14.2.3　滤波器的级联

假设有两个滤波器 h1 和 h2，将 h1 的输出信号输入到 h2，这样得到的滤波器称为 h1 和 h2 的级联。级联后的滤波器的脉冲响应为 h1 和 h2 的脉冲响应的卷积，频率响应为两个滤波器的频率响应的乘积。

下面的程序先用 remez()分别设计一个高通滤波器 h1 和一个低通滤波器 h2，然后通过卷积计算出它们的级联滤波器 h3 的系数。最后使用 freqz()计算 h3 的频率响应，结果如图 14-10 所示，可以看出滤波器 h3 是一个带通滤波器。



filter_cascade.py
使用卷积将低通和高通滤波器级联

```
import scipy.signal as signal
import numpy as np
import pylab as pl

h1 = signal.remez(201, (0, 0.18,  0.2,  0.50), (0.01, 1))
h2 = signal.remez(201, (0, 0.38,  0.4,  0.50), (1, 0.01))
h3 = np.convolve(h1, h2)

w, h = signal.freqz(h3, 1)
pl.plot(w/2/np.pi, 20*np.log10(np.abs(h)))

pl.legend()
pl.xlabel(u"正规化频率 周期/取样")
pl.ylabel(u"幅值(dB)")
pl.show()
```

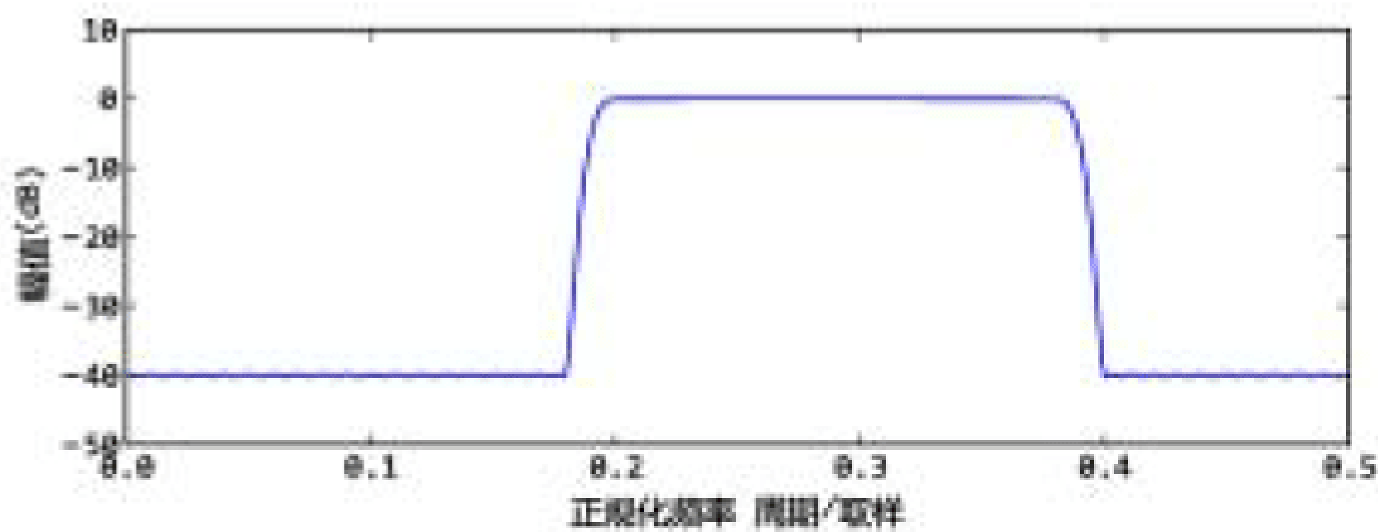


图 14-10　将低通、高通滤波器级联，得到带通滤波器

也可以直接用 remez()设计此带通滤波器：

```
>>> h4 = signal.remez(201, (0, 0.18, 0.2, 0.38, 0.4, 0.50), (0.01, 1, 0.01))
```

如果观察此滤波器的频率响应，就会发现它和 h3 的基本一致，图 14-11 比较级联滤波器的系数和 remez()所设计的带通滤波器的系数(见文前彩插)。

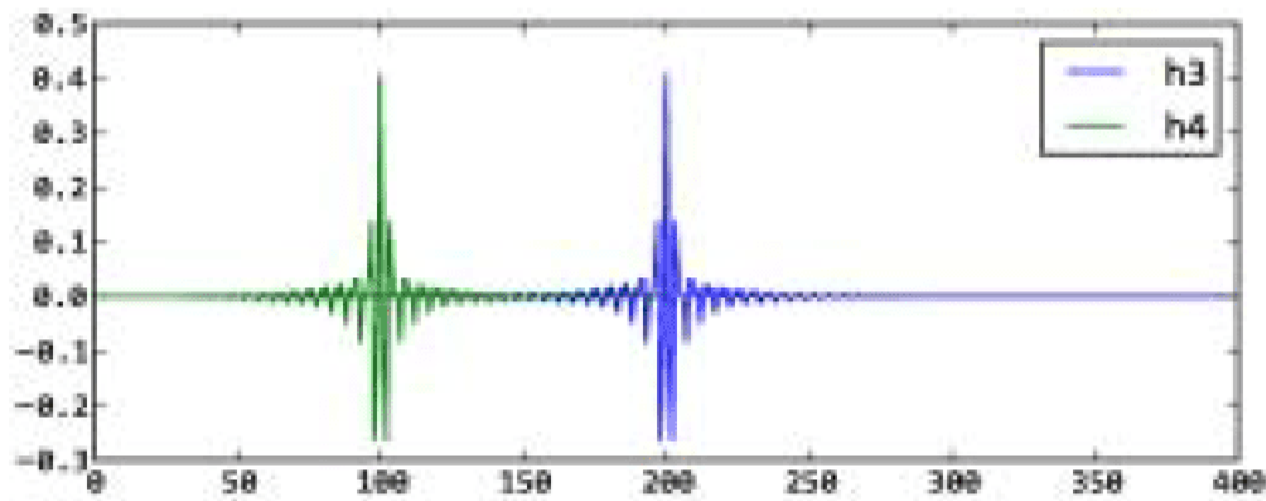


图 14-11　级联滤波器和由 remez()设计的带通滤波器的脉冲响应近似

可以看出，虽然 h3 的长度几乎是 h4 的两倍，但是由于它的许多系数都接近于 0，因此除了延时不同之外，h3 和 h4 的频率响应近似相同。

14.3 IIR 滤波器设计

在设计数字 IIR 滤波器时，通常都会先设计一个对应的模拟滤波器，然后通过双线性变换将模拟滤波器转换为数字滤波器。这意味着需要在 s 域中设计滤波器的传递函数 H(s)。当 H(s) 的所有极点都在 s 复平面的左半平面时，滤波器的响应是稳定的。下面以巴特沃斯低通滤波器为例说明这一设计过程。

14.3.1 巴特沃斯低通滤波器

巴特沃斯低通滤波器的振幅的平方与频率之间的关系可以用如下公式表示：

$$|H(j\omega)|^2 = \frac{1}{1+(\frac{\omega}{\omega_c})^{2n}}$$

其中，n 为滤波器的阶数，ω_c 为振幅下降 3dB 时的截止频率。这个公式很容易理解：

- 当 $\omega<\omega_c$ 时，ω 越小，振幅越接近于 1。
- 当 $\omega>\omega_c$ 时，ω 越大，振幅越接近于 0。
- 随着 n 的增大，振幅接近于 1 或 0 的速度将变快。即 n 越大，低通滤波器在阻频带的衰减速度将越快。
- 当 $\omega=\omega_c$ 时，振幅的平方为 1/2，即-3dB。

下面我们推导出巴特沃斯低通滤波器的传递函数 H(s)，其中 $s=\delta+j\omega$，是复数平面上的点。

由于当 $s=j\omega$ 时，$H(s)H(-s)=|H(j\omega)|^2$，因此将 $\omega=s/j$ 带入到巴特沃斯低通滤波器的公式中，可以得到：

$$H(s)H(-s)=\frac{1}{1+(\frac{-s^2}{\omega_c^2})^n}$$

此公式有 2n 个极点，其中 n 个在左半复平面，n 个在右半复平面，由于 *H(s)* 必须是稳定的，因此左半平面的 n 的极点属于 *H(s)*。

最后得到的传递函数为：

$$H(s)=\frac{1}{\prod_{k=1}^{n}(s-s_k)/\omega_c}$$

其中 s_k 为左半平面上的极点：

$$s_k=\omega_c e^{\frac{j(2k+n-1)\pi}{2n}} \quad k=1,2,3,\ldots,n$$

下面的程序绘制 6、7 阶巴特沃斯低通滤波器的 s 复平面上的极点：



filter_butter.py
巴特沃斯低通滤波器的极点

```
from scipy import signal
import numpy as np
import matplotlib.pyplot as pl

pl.figure(figsize=(6,6))

b, a = signal.butter(6, 1.0, analog=1)
z,p,k = signal.tf2zpk(b, a)
pl.plot(np.real(p), np.imag(p), '^', label=u"6 阶")

b, a = signal.butter(7, 1.0, analog=1)
z,p,k = signal.tf2zpk(b, a)
pl.plot(np.real(p), np.imag(p), 's', label=u"7 阶")

pl.axis("equal")
pl.legend(loc="center right")
pl.show()
```

程序中，使用 butter()设计巴特沃斯低通滤波器，默认情况下设计的是数字滤波器，为了设计模拟滤波器，需要设置 analog 参数为 1。获得传递函数的系数 b 和 a 之后，通过 tf2zpk()将它们转换为零点和极点，结果如图 14-12 所示。

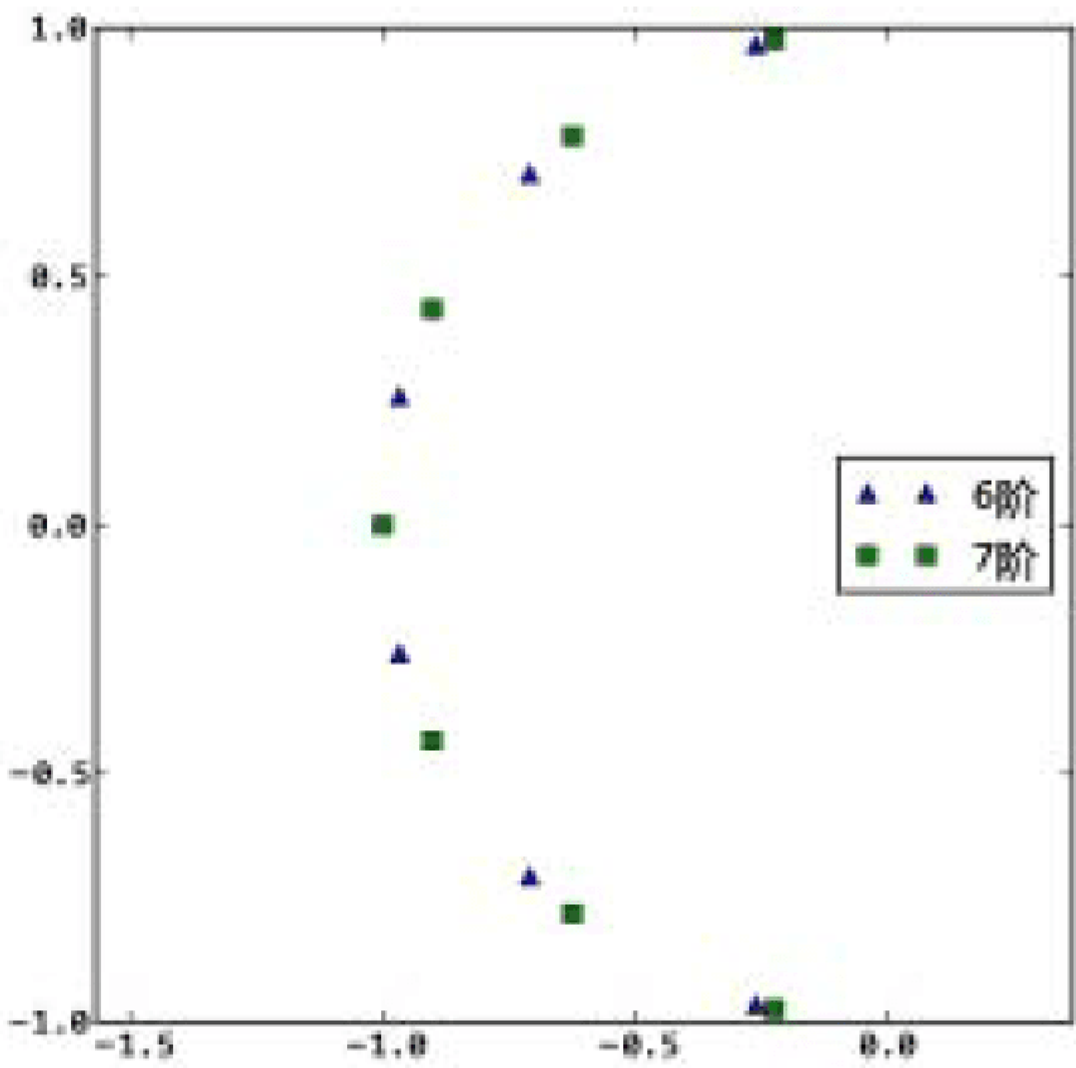


图 14-12　巴特沃斯低通滤波器在 s 复平面上的极点分布

注意，当 analog 参数为 1 时，butter()得到的系数是传递函数 $H(s)$的系数，而不是前面介绍的数字 IIR 滤波器的系数。假设 b 和 a 的长度分别为 M 和 N，模拟滤波器的系数中 b[0]为分子中 s^{m-1} 项的系数，a[0]为分母中 s^{n-1} 项的系数，b[-1]和 a[-1]为分子分母的常数项，即：

$$H(s)=\frac{b_0 s^{M-1}+b_1 s^{M-2}+\cdots+b_{M-1}}{a_0 s^{N-1}+a_1 s^{N-2}+\cdots+a_{N-1}}$$

14.3.2 双线性变换

有了连续时间系统的传递函数 H(s)，下一步就是将它转换为离散时间系统的传递函数 H(z)。转换的方法有几种，其中最常用的是双线性变换，其变换公式为：

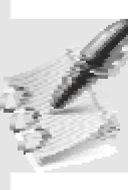
$$s \leftarrow \frac{2}{T}\frac{z-1}{z+1}$$

其中，T 为离散时间系统的取样周期。双线性变换公式的推导过程请参考下面的链接：



http://en.wikipedia.org/wiki/Bilinear_transform
双线性变换公式的推导

双线性变换实际上是 s 复平面和 z 复平面中点的映射变换，它将 s 复平面中的竖线变换成 z 复平面中的圆，而 s 复平面中的 Y 轴对应于 z 复平面中的单位圆。下面的程序演示了这一对应关系，结果如图 14-13 所示(见文前彩插)。



filter_bilinear_transform.py
双线性变换演示

```
import numpy as np
import pylab as pl

def stoz(s):
    """
    将 s 复平面映射到 z 复平面
    为了方便起见，假设取样周期 T=1
    """
    return (2+s)/(2-s)

def make_vline(x):
    return x + 1j*np.linspace(-100.0,100.0,20000)

fig = pl.figure(figsize=(7,3))
axs = pl.subplot(121)
axz = pl.subplot(122)
```

```
for x in np.arange(-3, 4, 1):
    s = make_vline(x)
    z = stoz(s)
    axs.plot(np.real(s), np.imag(s))
    axz.plot(np.real(z), np.imag(z))

axs.set_xlim(-4,4)
axz.axis("equal")
axz.set_ylim(-3,3)

pl.show()
```

程序中，stoz()将 s 复平面中的点变换为 z 复平面中的点，为了方便起见，这里假设取样周期 T 为 1。

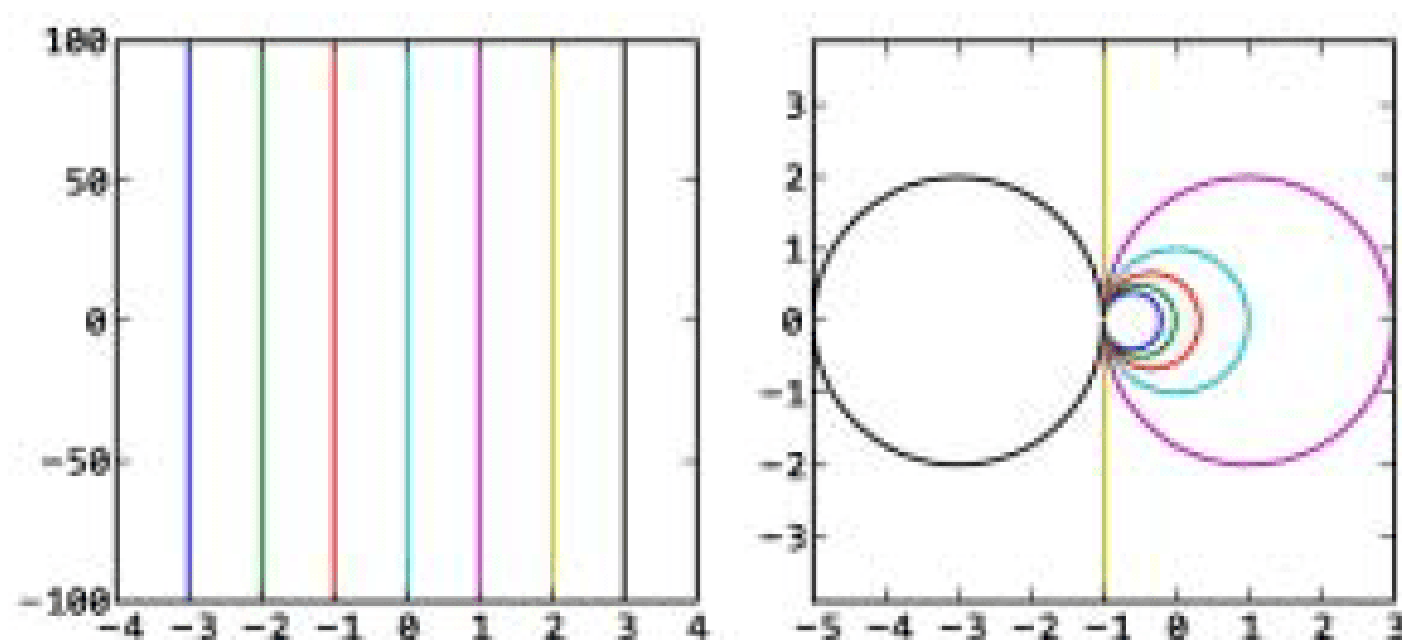


图 14-13 双线性变换将 s 复平面中的竖线(左图)变换为 z 复平面中的圆(右图)

进行双线性变换之后，滤波器的频率响应会发生变化。将离散时间系统的传递函数 *H(z)* 用 $z=e^{j\omega T}$ 进行替换，就可以得到其频率响应。将其带入双线性变换公式，得到：

$$s=\frac{2}{T}\frac{z-1}{z+1}=\frac{2}{T}\frac{e^{j\omega T}-1}{e^{j\omega T}+1}=\frac{2}{T}\frac{e^{j\omega T/2}-e^{-j\omega T/2}}{e^{j\omega T/2}+e^{-j\omega T/2}}=j\frac{2}{T}\tan(\omega T/2)$$

对于 s 复平面上的点来说，过原点的竖线 $s=j\omega$ 就是连续时间系统的传递函数 H(s)的频率响应，因此双线性变换将离散时间系统的频率 ω 通过如下公式转换为连续时间系统的频率 ω_a：

$$\omega_a=\frac{2}{T}\tan(\omega T/2)$$

其反函数为：

$$\omega=\frac{2}{T}\arctan(\omega_a T/2)$$

下面我们用程序验证这个频率转换公式。首先载入所需要的库，并且定义离散时间系统的取样频率 fs 为 8k Hz，所设计的巴特沃斯低通滤波器的通带截止频率 fc 为 1k Hz：



filter_bilinear_freq.py
验证双线性变换的频率转换公式

```
>>> from scipy import signal
>>> from numpy import *
>>> fs = 8000.0
>>> fc = 1000.0
```

下面使用 butter()设计一个 3 阶的巴特沃斯低通滤波器。注意，analog 参数为 1，表示设计连续时间系统的传递函数 H(s)的系数。由于通带截止频率参数为圆频率，因此需要将通带截止频率 fc 乘以 2π：

```
>>> b, a = signal.butter(3, 2*pi*fc, analog=1)
```

然后调用双线性变换函数 bilinear()，将传递函数转换为离散时间系统，通过 fs 参数指定取样频率：

```
>>> b2, a2 = signal.bilinear(b,a,fs=fs)
```

接下来用 freqz()得到此数字滤波器的频率响应，为了得到尽可能精确的值，我们通过 worN 参数设置它计算 10000 点的频率响应：

```
>>> w2, h2 = signal.freqz(b2,a2,worN=10000)
```

接下来将 h2 转换为增益，并且找到增益为-3dB(精确值为 10*log10(0.5))时对应的正规化圆频率 w 的下标 idx，w/(2*pi)*fs 就是其对应的实际频率值：

```
>>> p2 = 20*log10(abs(h2))
>>> idx = argmin(abs(p2-10*log10(0.5)))
>>> w2[idx]/2/pi*fs
952.8
```

通过频率转换公式得到的离散时间系统的截止频率为：

```
>>> 2*fs*arctan(2*pi*fc/2/fs) /2/pi
952.8840223
```

实际使用 signal 库设计 IIR 滤波器时没有这么麻烦，因为它所提供的滤波器设计函数默认都是直接设计数字滤波器。这些函数在设计数字滤波器时采用的取样频率为 2，即以香农频率 fs/2 为 1 进行正规化。因此要设计取样频率为 fs、通带频率为 f 的滤波器，需要将通带频率正规化为 f/(fs/2)。下面调用 butter()设计数字低通滤波器，这里使用前面计算得到的通带频率：

```
>>>  b3,a3 = signal.butter(3, 952.8840223/(fs/2))
```

```
>>> sum(abs(b3-b2))
1.3226225670237568e-13
>>> sum(abs(a3-a2))
7.0876637892069994e-13
```

前面计算得到的滤波器的系数 b3 和 a3 与通过 bilinear()计算的系数 b2 和 a2 是一致的。在使用 signal 库设计数字滤波器时，其内部会先通过频率转换公式对频率进行转换，然后设计传递函数的系数，最后通过 bilinear()进行系数转换。有兴趣的读者可以查看 iirfilter()的源代码。

14.3.3 滤波器的频带转换

只要知道了低通滤波器的传递函数 $H(s)$，就很容易利用变量替换法设计出同样阶数的高通、带通或其他通带频率的低通滤波器。

假设采用巴特沃斯低通滤波器的设计公式，设计出如下通带频率为 1 弧度/秒的标准低通滤波器：

```
>>> b, a = signal.butter(2, 1.0, analog=1)
>>> np.real(b)
array([ 1.])
>>> np.real(a)
array([ 1.        ,  1.41421356,  1.        ])
```

此滤波器的传递函数如下：

$$H(s) = \frac{1}{s^2 + 1.4142s + 1}$$

为了让它变为通带频率为 ω_c 的低通滤波器，只需要进行如下变量替换：

$$s \rightarrow \frac{s}{\omega_c}$$

由于将 $s=j\omega$ 代入传递函数 $H(s)$就能得到滤波器的频率响应，因此上面设计的标准低通滤波器 H(s)在 $\omega=1$ 时的增益为-3dB。而 $H(s/\omega_c)$在 $\omega=\omega_c$ 处的增益为-3dB。下面的语句计算通带频率为 2 弧度/秒的低通滤波器的系数：

```
>>> b2, a2 = signal.butter(2, 2.0, analog=1)
>>> np.real(b2)
array([ 4.])
>>> np.real(a2)
array([ 1.        ,  2.82842712,  4.        ])
```

可以看出，将 $s \rightarrow \frac{s}{2}$ 代入到标准低通滤波器的传递函数中即可得到这些系数。

低通滤波器转高通滤波器的变量替换公式为：

$$s \to \frac{\omega_c}{s}$$

此公式也很容易理解：

- 若 ω 为 0，则替代之后的频率为无穷大，而低通滤波器在频率为无穷大时的响应为 0，即转换之后的滤波器在 0 处的频率响应为 0。
- 若 ω 为无穷大，则替代之后的频率为 0，因此转换之后的滤波器在频率为无穷大时的响应为 1。

下面设计通带频率为 1 弧度/秒的高通滤波器：

```
>>> b3,a3 = signal.butter(2,1.0,btype="high",analog=1)
>>> np.real(b3)
array([ 1.,  0.,  0.])
>>> np.real(a3)
array([ 1.        ,  1.41421356,  1.        ])
```

可以看出，这些系数是将 $s \to \frac{1}{s}$ 代入到 $H(s)$ 中之后，分子和分母分别乘以 s^2 得到的。

低通滤波器还可以转换为带通滤波器。这可能有点难以理解，我们先看看变量替换公式。假设带通滤波器的高低通带频率为 ω_2 和 ω_1：

$$s \to \frac{\omega_0}{\Delta\omega}\left(\frac{s}{\omega_0} + \frac{\omega_0}{s}\right)$$

其中，$\Delta\omega = \omega_2 - \omega_1$，$\omega_0 = \sqrt{\omega_1\omega_2}$。$\Delta\omega$ 被称为通带带宽，ω_0 则是通带的中心频率。

我们通过下面的程序演示低通滤波器是如何转换成带通滤波器的：



Filter_bandpass_iir.py
低通滤波器转带通滤波器的原理演示

```
import numpy as np
from scipy import signal
import pylab as pl

b, a = signal.butter(2, 1.0, analog=1) ❶

# 低通->带通的频率变换函数
w1 = 1.0 # 低通带频率
w2 = 2.0 # 高通带频率
```

```
dw = w2 - w1 # 通带宽度
w0 = np.sqrt(w1*w2) # 通带中心频率

# 产生 10**-2 到 10**2 的频率点
w = np.logspace(-2, 2, 1000) ❷

# 使用频率变换公式计算出转换之后的频率
nw = np.imag(w0/dw*(1j*w/w0 + w0/(1j*w))) ❸

_, h = signal.freqs(b, a, worN=nw) ❹
h = 20*np.log10(np.abs(h))

pl.figure(figsize=(8,5))

pl.subplot(221)
pl.semilogx(w, nw) # X 轴使用 log 坐标绘图
pl.xlabel(u"变换前圆频率(弧度/秒)")
pl.ylabel(u"变换后圆频率(弧度/秒)")

pl.subplot(222)
pl.plot(h, nw)
pl.xlabel(u"低通滤波器的频率响应(dB)")

pl.subplot(212)
pl.semilogx(w, h) ❺
pl.xlabel(u"变换前圆频率(弧度/秒)")
pl.ylabel(u"带通滤波器的频率响应(dB)")

pl.subplots_adjust(wspace=0.3, hspace=0.3, top=0.95, bottom=0.14)

print "center:", w[np.argmin(np.abs(nw))]
pl.show()
```

❶先用 butter()设计一个模拟的二阶标准低通滤波器，将其转换为通带频率为 w1=1 到 w2=2 的带通滤波器。❷使用 logspace()产生频率段为 0.01 到 100 的等比数列 w，此后的频率响应运算都针对 w 进行。

❸通过带通频率转换公式将其转换为新的频率序列 nw。❹并使用 nw 计算每个频率点对应的低通滤波器的频率响应，我们通过 freqs()的 worN 参数指定需要计算频率响应的频率点。

❺用 semilogx()将 h 和 w 的关系绘制成如图 14-14(下)所示的对数坐标图，即带通滤波器的频率响应。左上图绘制的是频率转换函数，右上图绘制的是低通滤波器的频率响应(X 轴为响应，Y 轴为频率)。

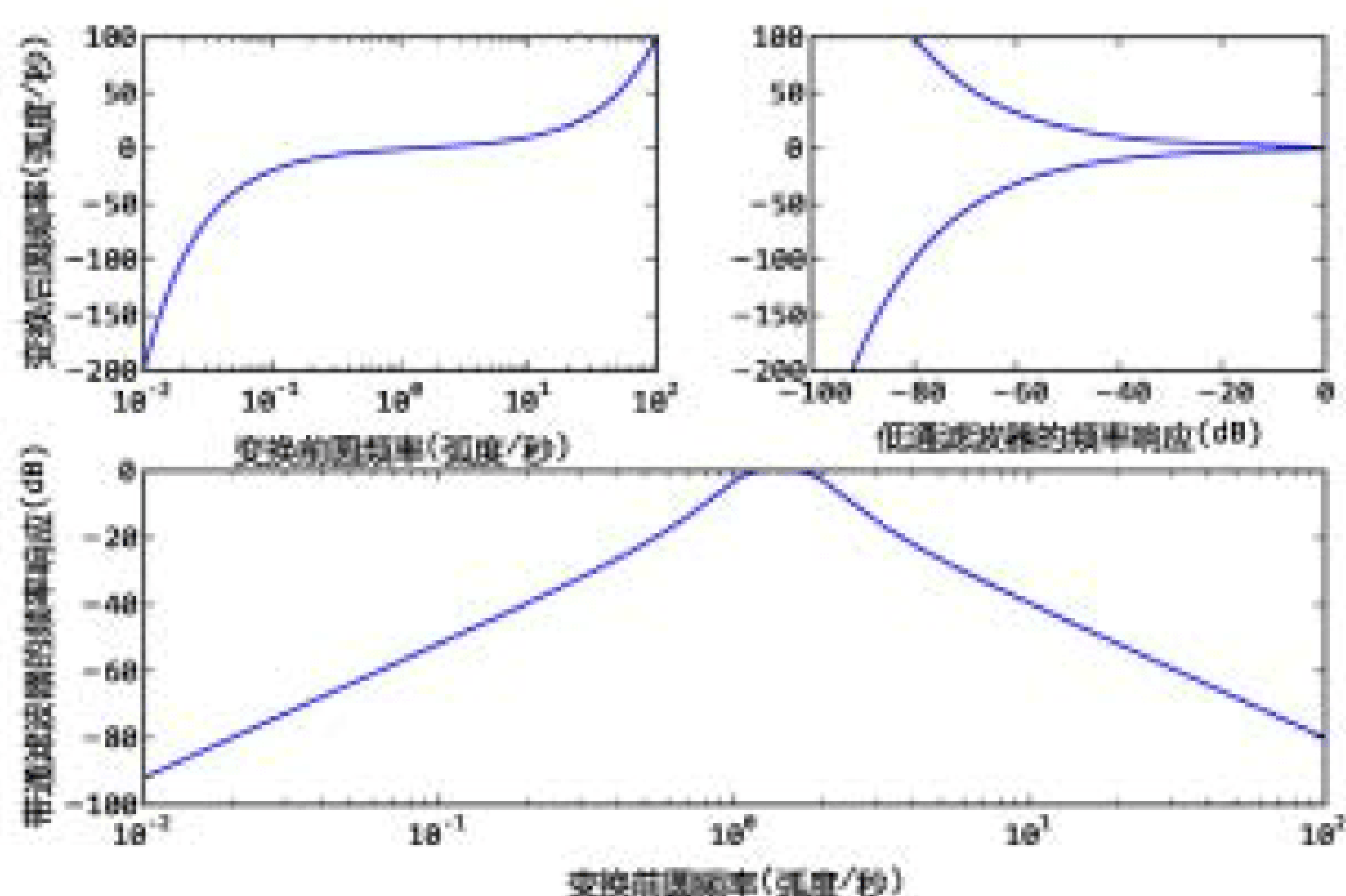


图 14-14　使用频率转换公式将低通滤波器变为带通滤波器

由于带通滤波器的频率转换公式将 0 到无穷大映射到负无穷到正无穷，而低通滤波器在正负无穷处的频率响应都为 0，因此可以想象转换后的滤波器是一个带通滤波器。而中心频率 ω_0 经公式转换之后的值为 0，低通滤波器在 0 处的频率响应为 1，因此带通滤波器在 ω_0 处的频率响应为 1。

事实上，signal 库已经提供了频带转换函数：

- lp2lp()：低通转低通。
- lp2hp()：低通转高通。
- lp2bp()：低通转带通。
- lp2bs()：低通转带阻。

下面以 lp2bp()为例简要说明函数的用法。假设 b、a 为二阶标准低通滤波器，下面的程序将其转换为通带为 1 到 2 的带通滤波器。lp2bp()的前两个参数为滤波器的系数，后两个参数分别为中心频率和通带带宽：

```
>>> b, a = signal.butter(2, 1.0, analog=1)
>>> b3, a3 = signal.lp2bp(b,a,np.sqrt(2), 1)
```

也可以直接调用 butter()设计一个低通滤波器：

```
>>> b4, a4 = signal.butter(2, [1,2], btype='bandpass', analog=1)
```

二者的结果是完全一致的：

```
>>> np.all(b3==b4)
True
```

```
>>> np.all(a3==a4)
True
```

14.4 数字滤波器的频率响应

前面的许多例子都是使用 freqz()计算数字滤波器的频率响应，本节将深入研究 freqz()是如何计算频率响应的。在 IPython 中输入：

```
>>> import scipy.signal
>>> signal.freqz??
```

即可看到 freqz()的代码，下面是其完整的源程序：

```
def freqz(b, a=1, worN=None, whole=0, plot=None):
    b, a = map(atleast_1d, (b,a))
    if whole:
        lastpoint = 2*pi
    else:
        lastpoint = pi
    if worN is None:
        N = 512
        w = numpy.arange(0,lastpoint,lastpoint/N)
    elif isinstance(worN, types.IntType):
        N = worN
        w = numpy.arange(0,lastpoint,lastpoint/N)
    else:
    w = worN
    w = atleast_1d(w)
    zm1 = exp(-1j*w)
    h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
    if not plot is None:
        plot(w, h)
    return w, h
```

研究这段代码，不难发现真正计算频率响应的代码可以用如下 3 行代码概括：

```
w = numpy.arange(0,pi,pi/N)
zm1 = exp(-1j*w)
h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
```

为了弄清楚为什么这 3 行代码能够计算数字滤波器的频率响应，让我们先来学习一下相关

的理论知识。

数字滤波器的频率响应由滤波器的传递函数给出，IIR 滤波器的计算公式如下：

$$y[m] = b[0]x[m] + b[1]x[m-1] + \cdots + b[P]x[m-P] \\ - a[1]y[m-1] - a[2]y[m-2] - \cdots - a[Q]y[m-Q]$$

根据 Z 变换的相关公式，很容易求得其传递函数为：

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b[0] + z^{-1}b[1] + z^{-2}b[2] + \cdots + z^{-M}b[P]}{1 + z^{-1}a[1] + z^{-2}a[2] + \cdots + z^{-N}a[Q]}$$

其中，z 为复数平面上的任意一点。当 z 为单位圆上的点，即 $z = e^{j\omega}$ 时，$H(\omega)$ 就是滤波器的频率响应。ω 被称为圆频率，当其取值从 0 到 2π 变化时，$e^{j\omega}$ 正好绕复数平面单位圆转一圈。由于复数平面上下两个半平面的复数存在共轭关系，因此通常只需要求得上半圆的频率响应即可。下面的语句将上半圆等分为 N 份：

```
w = numpy.arange(0,pi,pi/N)
```

然后计算 w 中每点对应的复数值 $e^{-j\omega}$：

```
zm1 = exp(-1j*w)
```

于是，传递函数的分子分母部分就都变成了 zm1 的多项式函数。最后将 zm1 代入传递函数的公式中，计算出频率响应 h：

```
h = polyval(b[::-1], zm1) / polyval(a[::-1], zm1)
```

polyval(p, x)对数组 x 中的每个元素计算多项式 p 的值，其计算公式如下：

```
p[0]*(x**N-1) + p[1]*(x**N-2) + ... + p[N-2]*x + p[N-1]
```

由于滤波器系数 b 和 a 的顺序正好和 polyval()的多项式系数 p 的顺序相反，因此通过数组切片运算 b[::-1]将滤波器的系数反转。由于数组 zm1 中的值都为复数，因此得到的频率响应 h 的值也都是复数。复数的幅值对应于频率响应中的增益特性，相角对应于频率响应中的相位特性。

我们经常需要在绘制频率响应图表时要求频率坐标为对数坐标。对于对数坐标，arange()创建的等距的频率数组 w 会造成低频过疏、高频过密的问题。为了解决这个问题，可以先用 logspace()计算一个等比的频率数组，并通过 worN 参数将其传递给 freqz()。当 worN 参数是一个序列对象时，freqz()对其中指定的频率计算响应。



filter_logfreqz.py
用 freqz()计算对数坐标的频率响应

```
import numpy as np
import pylab as pl
import scipy.signal as signal

def logfreqz(b, a, f0, f1, fs, N):
    """
    以对数频率坐标计算滤波器 b、a 的频率响应
    f0, f1: 计算频率响应的开始频率和结束频率
    fs: 取样频率
    """
    w0, w1 = np.log10(f0/fs*2*np.pi), np.log10(f1/fs*2*np.pi) ❶
    # 不包括结束频率
    w = np.logspace(w0, w1, N, endpoint=False) ❷
    _, h = signal.freqz(b, a, worN=w) ❸
    return w/2/np.pi*fs, h

for n in range(1, 6):
    # 设计 n 阶的通频为 0.1*4000 = 400 Hz 的高通滤波器
    b, a = signal.iirfilter(n, [0.1, 1])  ❹
    f, h = logfreqz(b, a, 10.0, 4000.0, 8000.0, 400)
    gain = 20*np.log10(np.abs(h))
    pl.semilogx(f, gain, label="N=%s" % n)
    slope = (gain[100]-gain[10]) / (np.log2(f[100]) - np.log2(f[10])) ❺
    print "N=%s, slope=%s dB" % (n, slope)
pl.ylim(-100, 20)
pl.xlabel(u"频率(Hz)")
pl.ylabel(u"增益(dB)")
pl.legend()
pl.show()
```

程序中，logfreqz()计算系数为 b 和 a 的滤波器在 f0 到 f1 之间的频率响应，f_s为取样频率，N 为计算的点数。❶首先通过$2\pi \cdot f / f_s$将实际频率转换为与之对应的圆频率。❷然后通过 logspace()计算频率点的等比数列。❸最后调用 freqz()计算等比数列中每点的频率响应，返回值为以 Hz 为单位的频率及其对应的频率响应。

❹调用 iirfilter()设计 5 个不同阶数的 IIR 高通滤波器，通频为 0.1。如果取样频率为 8 kHz，那么实际的通频为 0.1*4k Hz=400 Hz。5 个 IIR 滤波器的增益特性如图 14-15 所示(见文前彩插)。

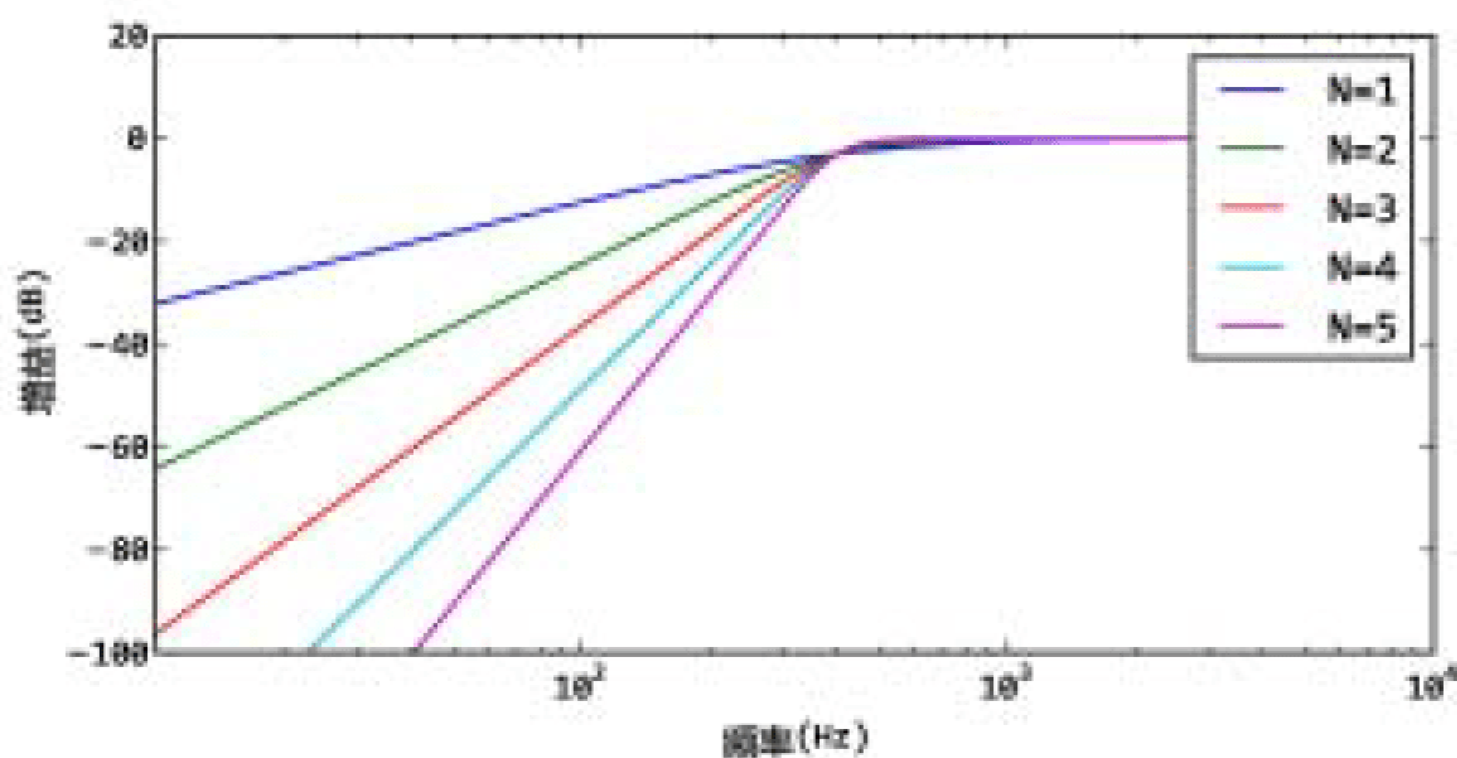


图 14-15　使用 iirfilter()设计 5 个不同阶的 IIR 高通滤波器

由图 14-15 可知，随着 IIR 滤波器阶数的增加，增益的下降速度也在增加。❺计算增益下降时倍频之间的增益差值，结果如下：

```
N=1, slope=5.9955865774 dB
N=2, slope=12.0417201051 dB
N=3, slope=18.0630802032 dB
N=4, slope=24.0841135443 dB
N=5, slope=30.1051375912 dB
```

即 IIR 滤波器的阶数每增加 1，增益的下降速度增加 6dB/oct (6dB 每倍频)，并且所有曲线相交于一点——此处的频率正好是 400 Hz，增益为-3 dB。

14.5　二次均衡滤波器设计工具

无论是古老的盒式录音机还是现代的流行数码音响设备，抑或众多的音乐播放软件，其中绝大多数的均衡器都只是由一系列简单的二次 IIR 滤波器组合而成。

二次 IIR 滤波器的传递函数如下：

$$H(z)=\frac{b0+b1\cdot z^{-1}+b2\cdot z^{-2}}{a0+a1\cdot z^{-1}+a2\cdot z^{-2}}$$

当 a0 的值不为 1 时，可以将所有系数(b0、b1、b2、a0、a1、a2)都除以 a0，这样就能得到 a0=1 时的 5 个系数——b0、b1、b2、a1、a2，即二次均衡滤波器的频率响应曲线由这 5 个独立的参数决定，其频率响应如图 14-16 所示(见文前彩插)。

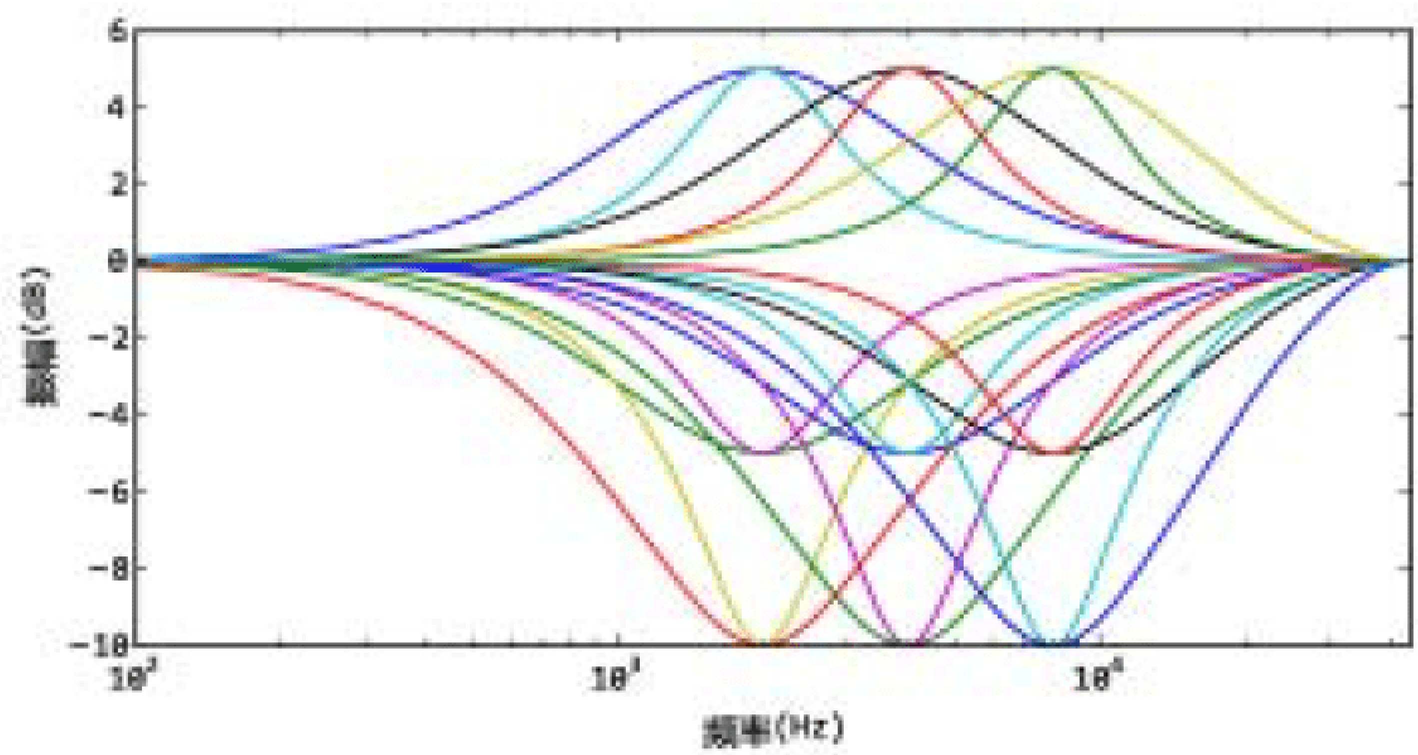


图 14-16　二次均衡滤波器的频率响应

从图 14-16 中可以看出频率响应曲线的两个参数——中心频率 f0 和振幅峰值 peak：

- 频率为 0 时，频率响应为 1。
- 频率为取样频率一半时(圆频率为 π)，频率响应为 1。
- 频率为中心频率 f0 时，振幅达到峰值 peak。
- 频率响应在中心频率 f0 处的导数为 0。
- 这样就有了 4 个方程，再加上一个 Q 值决定振幅驼峰的宽度，因此一共 5 个方程决定 5 个系数。

Audio EQ Cookbook 中提供了二次均衡滤波器系数的设计公式，根据这个设计手册，很容易写出如下设计二次均衡滤波器参数的函数：

http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt
二次 IIR 滤波器参数设计大全——Audio EQ Cookbook

filter_equalizer.py
设计二次均衡滤波器的参数

```
import scipy.signal as signal
import pylab as pl
import math
import numpy as np

def design_equalizer(freq, Q, gain, Fs):
    '''设计二次均衡滤波器的系数'''
    A = 10**(gain/40.0)
    w0 = 2*math.pi*freq/Fs
    alpha = math.sin(w0) / 2 / Q
```

```
    b0 = 1 + alpha * A
    b1 = -2*math.cos(w0)
    b2 = 1 - alpha * A
    a0 = 1 + alpha / A
    a1 = -2*math.cos(w0)
    a2 = 1 - alpha / A
    return [b0/a0,b1/a0,b2/a0], [1.0, a1/a0, a2/a0]

pl.figure(figsize=(8,4))
for freq in [1000, 2000, 4000]:
    for q in [0.5, 1.0]:
        for p in [5, -5, -10]:
            b,a = design_equalizer(freq, q, p, 44100)
            w, h = signal.freqz(b, a)
            pl.semilogx(w/np.pi*44100, 20*np.log10(np.abs(h)))
pl.xlim(100, 44100)
pl.xlabel(u"频率(Hz)")
pl.ylabel(u"振幅(dB)")
pl.subplots_adjust(bottom=0.15)
pl.show()
```

使用前面介绍的对数频率响应的求法以及 TraitsUI 和 Chaco 等界面库，我们可以设计如图 14-17 所示的二次均衡滤波器设计程序。



filter_equalizer_designer.py
二次均衡滤波器设计器

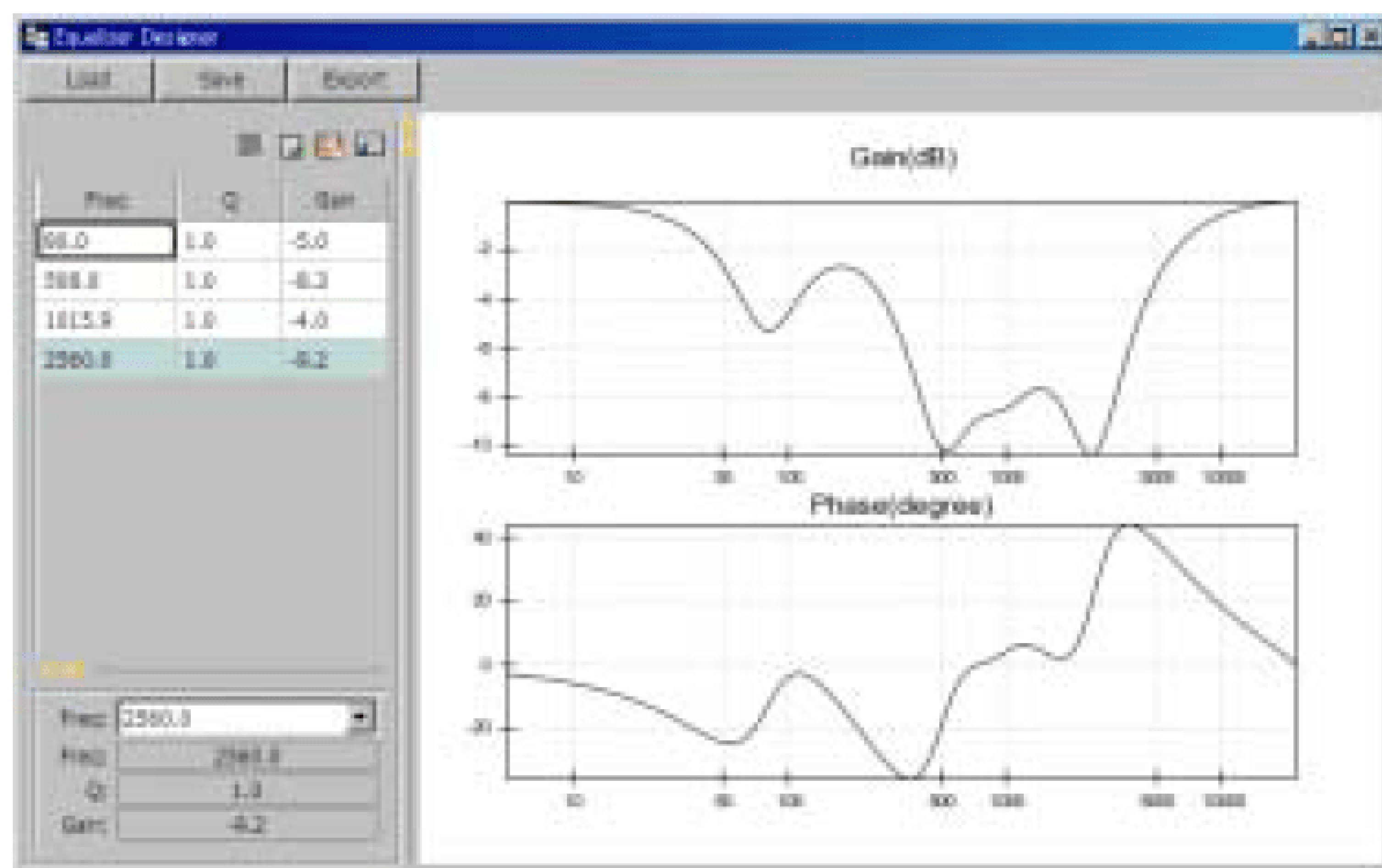


图 14-17　二次均衡滤波器设计工具的界面

用户可以使用此程序添加、删除和编辑二次均衡滤波器，并且可以即时查看均衡器级联之后的频率响应，还可以将设计好的滤波器系数输出为如下的 C 语言文件：

```
double EQ_PARS[][5] = {
//b0,b1,b2,a1,a2 // frequency, Q, gain
    {1.00073497929,-1.97887290381,0.978711709005,
                  -1.97887290381,0.979446688291}, // 169.0,1.12,0.6
};
```

14.6　零相位滤波器

使用 signal 库的 lfilter()对数据进行滤波时，会产生相位偏移，这是因为实时的信号处理滤波器在计算 t 时刻的滤波器的输出时，只能使用之前的输入和输出信号，否则就不满足因果律。然而很多情况下，我们可以先将输入信号完全记录下来，然后再进行处理，这时就可以使用不满足因果律的滤波函数 filtfilt()进行滤波，它能够保证输出信号和输入信号的相位完全一致。下面的程序演示了 filtfilt()的滤波结果：



filter_filtfilt.py
零相位滤波器演示

```
import numpy as np
from scipy import signal
import pylab as pl

t = np.arange(-1, 1, 0.01)
x = np.sin(np.pi*t+2) + np.random.randn(len(t))*0.05

[b,a] = signal.butter(3, 0.05)
z = signal.lfilter(b, a, x)
y = signal.filtfilt(b, a, x)

pl.figure(figsize=(8,4))
pl.plot(x, label=u"原始数据")
pl.plot(z, label=u"lfilter 滤波")
pl.plot(y, label=u"filtfilt 滤波")
pl.legend()
pl.show()
```

程序中，我们首先创建一个带噪声的正弦波信号 x，然后用 butter()设计一个 3 阶的低通滤波器，分别使用 lfilter()和 filtfilt()对此信号进行滤波，得到的结果如图 14-18 所示(见文前彩插)。可以看出，lfilter()会产生一些延时，但 filtfilt()的结果与原信号完全同步。

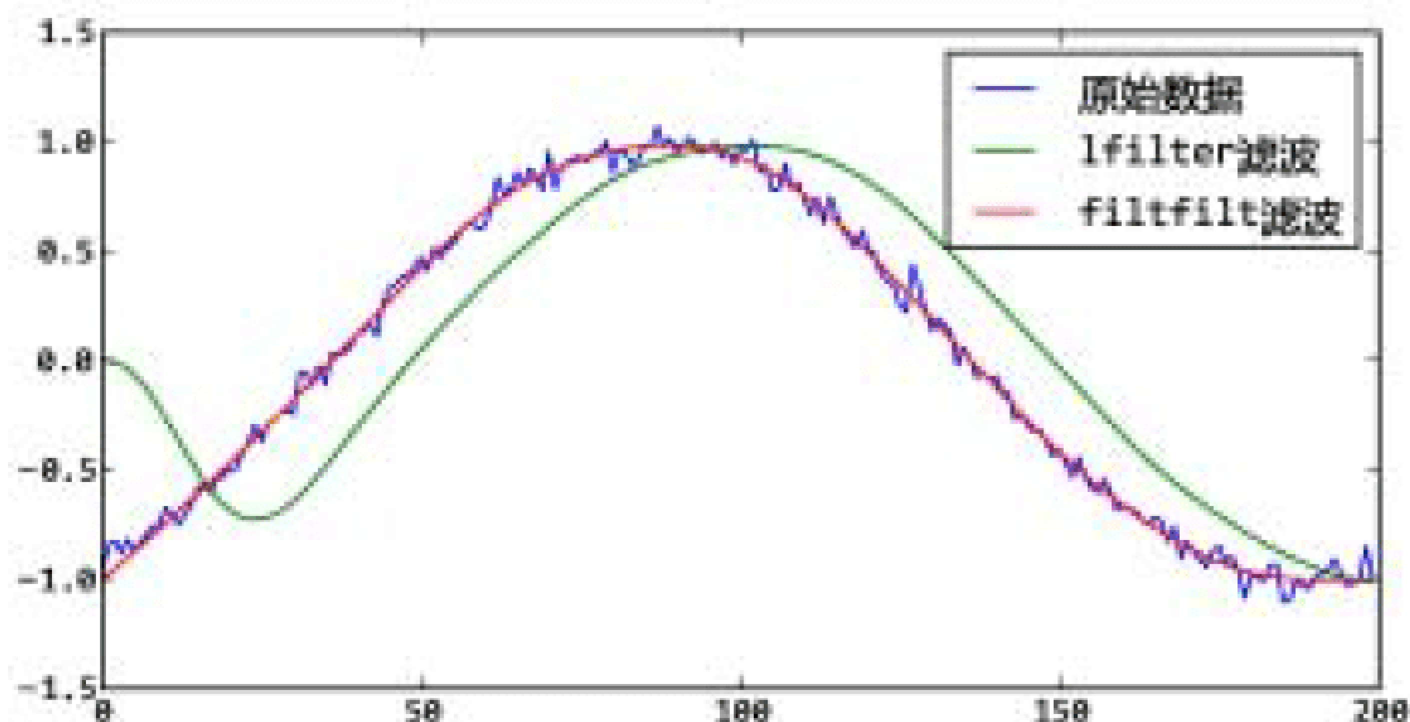


图 14-18　filtfilt()的滤波结果没有相位变化

14.7　重 取 样

把模拟信号转换为数字信号，需要对模拟信号按照一定的取样频率进行取样。对于声音信号，比较常用的取样频率有 8000 Hz、11025 Hz、16000 Hz、22050 Hz、32000 Hz、44100 Hz、48000 Hz 等，另外还有高取样频率 88200 Hz 和 96000 Hz 等。例如，CD 采用的是 44100 Hz，电话则一般采用 8000 Hz，MP3 文件格式则支持 48000 Hz 以下所有的取样频率。

假设要设计 MP3 播放器，为了能够播放所有取样频率，可以采用两种办法：一种是硬件支持，这需要数模转换器支持多种取样频率，硬件必须能够产生两套时钟——8000 Hz 和 11025 Hz，这无疑增加了硬件成本。第二种办法就是硬件只支持一种取样频率，通过软件改变声音信号的取样频率。

改变取样频率的基本方法就是对离散的数字信号进行插值。插值运算有多种算法，例如拉格朗日插值法、三阶样条曲线、贝赛尔曲线、线性插值，等等。这些算法在图形学领域应用十分广泛，但是它们不能用来对数字声音信号进行插值运算。

对于数字声音信号，我们关心的是插值运算的结果和原声音信号的误差。由香农定理可知，如果取样频率高于模拟信号的最高频率的两倍，就可以从取样后的数字信号精确地恢复原模拟信号。因此最理想的数字声音信号的插值算法就是基于此理论，从而精确地恢复原信号。得到原信号之后，只要再对其用新的取样频率进行取样，就可以实现重取样了。这种插值算法被称为限频带插值(Bandlimited Interpolation)。



http://ccrma.stanford.edu/~jos/resample
限频带插值的原理

假设对于连续时间信号 $x(t)$进行取样之后得到一个数列 $x(nT_s)$，这里的 t 表示连续时间，n 是一个整数，而 T_s 是取样周期。假设信号 $x(t)$的频带在 $-F_s/2$ 和 $F_s/2$ 之间，F_s 是取样频率，即

$F_s=1/T_s$。

使用下面的公式可以将取样数列 $x(nT_s)$还原为原始的连续时间信号 $x(t)$：

$$x(t)=\sum_{n=-\infty}^{\infty}x(nT_s)h(t-nT_s)$$

$$h(t)=\mathrm{sinc}(F_s t)=\frac{\sin(\pi F_s t)}{\pi F_s t}$$

分析一下此公式的含义。取样之后的数字信号可以看做原模拟信号与周期是 T_s 的脉冲信号的乘积。由于两个时域信号的乘积，等于它们的频域信号的卷积，因此取样之后信号的频域就是原信号频谱与周期脉冲信号的频谱的卷积。脉冲信号转换成频域之后仍然是脉冲，两个脉冲之间的间隔刚好是取样频率 F_s。这样一来，频域卷积的结果就是原模拟信号的频谱在频率轴上不断地重复，就好像时域中的周期信号一样。

为了将取样后的信号还原为原模拟信号，需要对其进行低通滤波，如果使用通带为$-F_s/2$到 $F_s/2$ 的理想低通滤波器，就可以精确地得到原信号的频谱。这种理想低通滤波器转换为时域波形，就是 sinc 函数。因此还原之后的模拟信号可以看做取样信号和 sinc 函数的卷积。或者可以理解为：在每个取样点处放置一个 sinc 函数波形，最终的模拟信号就是所有这些波形的叠加。如图 14-19 所示，数字信号的取样周期 T_s 为 1，4 个脉冲表示数字信号中的 4 个取样点，将其还原为模拟信号时，在每个取样点处放置一个用虚线表示的 sinc 函数波形，最后的模拟信号就是这 4 个波形的叠加，图中用粗实线表示。由于 sinc 函数波形在每个取样点处的值都为 0，因此得到的模拟信号正好经过所有的取样点。

filter_resample_sinc.py
绘制将数字信号还原为模拟信号的示意图

虽然理论上可以把取样后的信号还原为原信号，但是由于 sinc 函数波形是无限长的，实际计算中，只能取 sinc 函数波形的一部分进行卷积计算，并且需要使用窗函数加快收敛速度。

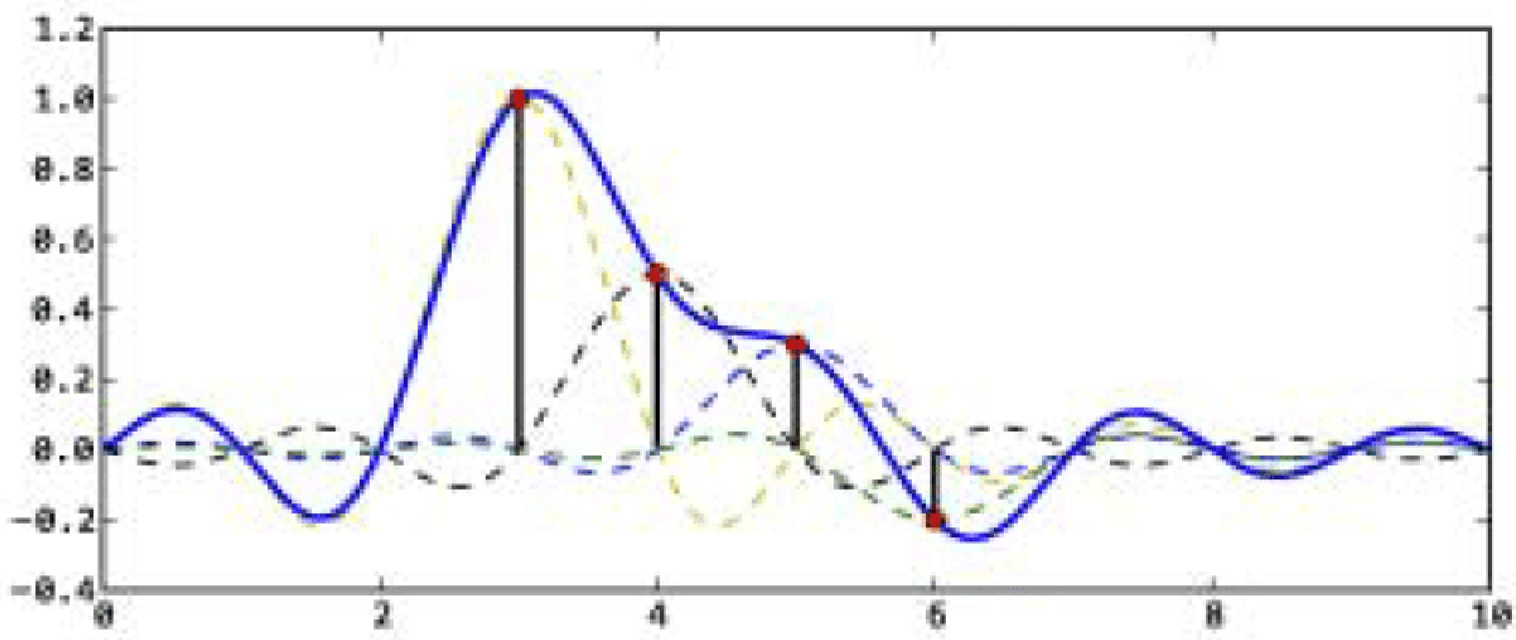


图 14-19 使用 sinc 函数将数字信号还原为模拟信号



filter_resample_demo.py
限频带插值算法演示

下面我们用程序验证限频带插值算法。首先计算 100 点取样频率为 1 Hz、频率为 $1/2\pi$ 的正弦波：

```
>>> t = np.arange(0, 100)
>>> x = np.sin(t)
```

由于计算机无法计算真正的模拟信号，因此下面的程序计算取样频率提高 N 倍之后的数字信号。先计算理想的信号 x2：

```
>>> N = 100
>>> t2 = np.arange(0, 100, 1.0/N)
>>> x2 = np.sin(t2)
```

为了和 sinc 函数进行卷积，先将原始的数字信号 x 的取样频率提高 N 倍，x3 中的每 N 个取样点中有一个是 x 中的取样点：

```
>>> x3 = np.zeros(len(x)*N)
>>> x3[::N] = x
```

然后计算 sinc 函数波形与 Hanning 窗(汉宁窗，又称升余弦窗)乘积之后的波形：

```
>>> sinc = np.sinc(np.arange(-10, 10, 1.0/N))
>>> sincw = sinc * signal.hann(len(sinc))
```

分别用这两个 FIR 滤波器对信号 x3 进行低通滤波，得到信号 x4 和 x5：

```
>>> x4 = signal.lfilter(sinc, [1], x3)
>>> x5 = signal.lfilter(sincw, [1], x3)
```

为了和理想信号 x2 进行比较，移除掉因滤波带来的延时：

```
>>> x4 = x4[len(sinc)/2:]
>>> x5 = x5[len(sincw)/2:]
```

由于 $t<0$ 时输入波形为 0，因此滤波计算刚开始时会有较大的误差，我们只比较 x2 与 x4、x5 在 $t>10$ 之后的误差值。由下面的结果可知，x5 比 x4 更接近理想波形 x2。因此采用窗函数处理之后的 sinc 波形，能更精确地还原信号，这和图 14-7 中表示的结果是一致的。

```
>>> np.sum((x2[10*N:90*N] - x4[10*N:90*N])**2)
1.7983068796001929
>>> np.sum((x2[10*N:90*N] - x5[10*N:90*N])**2)
0.00013840982681864963
```

接下来请读者自己完成下面的工作：

- 将计算中出现的各种信号绘制成图。
- 用各阶 B 样条曲线插值对 x 进行插值，计算它们和 x2 之间的误差。

前面的计算步骤只是用来演示限频带插值算法的原理，实际的重取样计算可以使用 scikits.samplerate 库。



http://pypi.python.org/pypi/scikits.samplerate/0.3.1
scikits.samplerate 库的下载地址

安装好 scikits.samplerate 库之后，运行下面的代码，对信号 x 进行 100 倍取样频率的重取样：

```
>>> from scikits import samplerate
>>> x6 = samplerate.resample(x, 100, 'sinc_best')
>>> np.sum((x2[10*N:90*N] - x6[10*N:90*N])**2)
0.093249751451539964
```

其中，参数'sinc_best'设置重取样的算法和精度，它的候选值可以通过 available_convertors()获得：

```
>>> samplerate.available_convertors()
['sinc_medium', 'linear', 'sinc_fastest', 'zero_order_hold', 'sinc_best']
```

第 15 章

频域信号处理

用 FFT(快速傅立叶变换)能将时域的数字信号转换为频域信号。转换为频域信号之后，可以很方便地分析出信号的频率成分，在频域上进行处理，最终还可以将处理完毕的频域信号通过IFFT(快速傅立叶变换的逆变换)转换回时域信号，实现许多在时域无法完成的信号处理算法。本章将通过许多实例，介绍有关频域信号处理的一些基础知识。

15.1 FFT 演示程序

本节介绍如何使用 NumPy、Traits 以及 Chaco 等库，编写一个三角波的 FFT 演示程序。此程序演示了如何综合使用各种库，编写带界面的科学计算程序，并且可以帮助读者理解 FFT 的原理。

15.1.1 FFT 知识复习

在正式开始程序编写之前，首先让我们复习一下有关 FFT 变换的知识。

FFT 变换是针对一个数组的运算，数组的长度 N 通常是 2 的整数次幂，例如 64、128、256 等。数值可以是实数或复数，通常的时域信号都是实数，因此下面都以实数为例。我们可以把这一组实数想象成对某个连续信号按照一定取样周期进行取样而得，如果对有 N 个实数的数组进行 FFT 变换，将得到一个含有 N 个复数的数组，我们称此复数数组为频域信号，它的元素有如下规律：

- 下标为 0 和 N/2 的两个复数的虚数部分为 0。
- 下标为 i 和 N－i 的两个复数共轭，也就是其虚数部分数值相同、符号相反。

下面的例子演示了上述规律，先以 rand()随机产生含有 8 个元素的数组 x，然后用 fft()对其运算之后，观察到结果为 8 个复数，并且满足上面两条规律：

```
>>> x = np.random.rand(8)
>>> x
array([ 0.15562099,  0.56862756,  0.54371949,  0.06354358,  0.60678158,
        0.78360968,  0.90116887,  0.1588846 ])
>>> xf = np.fft.fft(x)
>>> xf
array([ 3.78195634+0.j        , -0.53575962+0.57688097j,
```

```
        -0.68248579-1.12980906j, -0.36656155-0.13801778j,
         0.63262552+0.j        , -0.36656155+0.13801778j,
        -0.68248579+1.12980906j, -0.53575962-0.57688097j])
```

FFT 变换的结果可以通过 IFFT 变换还原为原来的值：

```
>>> np.fft.ifft(xf)
array([ 0.15562099 +0.00000000e+00j,  0.56862756 +1.91940002e-16j,
        0.54371949 +1.24900090e-16j,  0.06354358 -2.33573365e-16j,
        0.60678158 +0.00000000e+00j,  0.78360968 +2.75206729e-16j,
        0.90116887 -1.24900090e-16j,  0.15888460 -2.33573365e-16j])
```

注意，ifft()的运算结果实际上和数组 x 是相同的，由于浮点数有运算误差，因此出现了一些非常小的虚数，如果有必要，可以调用 np.real()获取其中的实数部分。

FFT 变换和 IFFT 变换并没有增加或减少数据的个数：数组 x 中有 8 个实数数值，而数组 xf 中其实也只有 8 个有效的数值。

计算 FFT 结果中有用的数值

由于复数共轭和虚数部分为 0 等规律，真正有用的信息保存在下标从 0 到 N/2 的 N/2+1 个复数中，又由于下标为 0 和 N/2 的值虚数部分为 0，因此只有 N 个有效的数值。

下面看看 FFT 变换所得到的复数的含义：

- 下标为 0 的实数表示了时域信号中的直流成分。
- 下标为 i 的复数 a+b*j 表示时域信号中周期为 N/i 个取样值的正弦波和余弦波的成分，其中 a 表示余弦波的成分，b 表示正弦波的成分。

下面让我们通过几个例子验证一下，首先对一个直流信号进行 FFT 变换：

```
>>> x = np.ones(8)
>>> x
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> np.fft.fft(x)/len(x) # 为了计算各个成分的能量多少，需要将 FFT 的结果除以 FFT 的长度
array([ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j,
        0.+0.j])
```

所谓直流信号，就是其值不随时间变化，因此我们创建一个值全为 1 的数组 x，可以看到它的 FFT 结果除了下标为 0 的数值不为 0 以外，其余的都为 0。这表示时域信号是直流的，并且其能量为 1。

下面我们产生一个周期为 8 个取样的正弦波，然后观察其 FFT 结果：

```
>>> x = np.arange(0, 2*np.pi, 2*np.pi/8)
>>> y = np.sin(x)
```

```
>>> tmp = np.fft.fft(y)/len(y)
>>> print np.array_str(tmp, suppress_small=True)
[ 0.+0.j  -0.-0.5j  0.-0.j   0.-0.j   0.+0.j   0.-0.j   0.+0.j   0.+0.5j]
```

如何用 linspace 创建取样点

要计算周期为 8 个取样的正弦波，就需要把 0 到2π的区间等分为 8 份，如果直接用 linspace()，它产生的值为：

```
>>> np.linspace(0, 2*np.pi, 8)
array([ 0.        ,  0.8975979 ,  1.7951958 ,  2.6927937 ,  3.5903916 ,
    4.48798951,  5.38558741,  6.28318531])
>>> 2*np.pi / 0.8975979
7.0000000079986666
```

可以看出，程序只将 0 到2π的区间等分为 7 份。为了获得正确的结果，可以产生 9 个点，并设置 endpoint 参数为 False，最终结果将不包括最后的点：

```
>>> np.linspace(0, 2*np.pi, 9, endpoint=False)
array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925268 ,
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361])
```

为了便于观察结果，我们用 array_str()将数组转换字符串，并设置 suppress_small 参数为 True，将一些很小的数值显示为 0。现在观察正弦波的 FFT 计算结果：下标为 1 的复数的虚数部分为-0.5，而我们产生的正弦波的振幅为 1，它们之间的关系是-0.5*(-2)=1。接下来观察余弦信号的 FFT 结果：

```
>>> tmp = np.fft.fft(np.cos(x))/len(x)
>>> print np.array_str(tmp, suppress_small=True)
[-0.0+0.j  0.5-0.j  0.0+0.j  0.0+0.j  0.0+0.j -0.0+0.j  0.0+0.j  0.5-0.j]
```

只有下标为 1 的复数的实数部分为 0.5，和余弦波振幅之间的关系是 0.5*2=1。再看下面两个例子：

```
>>> tmp = np.fft.fft(2*np.sin(2*x))/len(x)
>>> print np.array_str(tmp, suppress_small=True)
[ 0.+0.j  0.+0.j -0.-1.j  0.-0.j  0.+0.j  0.+0.j -0.+1.j  0.-0.j]
>>> tmp = np.fft.fft(0.8*np.cos(2*x))/len(x)
>>> print np.array_str(tmp, suppress_small=True)
[-0.0+0.j -0.0+0.j  0.4-0.j  0.0-0.j  0.0+0.j  0.0-0.j  0.4+0.j -0.0+0.j]
```

上面产生的是周期为 4 个取样点的正弦和余弦信号，其 FFT 的有效成分在下标为 2 的复数中，其中正弦波的振幅为 2，其频域虚数部分的值为-1；余弦波的振幅为 0.8，频域中对应的值

为 0.4。

如果将两个同频率的正弦波和余弦波通过不同的系数进行叠加，就可以得到同样频率的各种相位的余弦波。因此我们可以这样来理解频域中的复数：

- 复数的模(绝对值)的两倍为对应频率的余弦波的振幅。
- 复数的辐角表示对应频率的余弦波的相位。

最后再看一个例子：

```
>>> x = np.arange(0, 2*np.pi, 2*np.pi/128)
>>> y = 0.3*np.cos(x) + 0.5*np.cos(2*x+np.pi/4) + 0.8*np.cos(3*x-np.pi/3)
>>> yf = np.fft.fft(y)/len(y)
>>> print np.array_str(yf[:4], suppress_small=True)
[ 0.0000000+0.j  0.1500000+0.j  0.1767767+0.1767767j  0.2000000-0.34641016j]
>>> np.abs(yf[1]), np.rad2deg(np.angle(yf[1])) # 周期为 128 取样点的余弦波的振幅和相位
(0.15000000000000008, 2.3980988870246962e-015)
>>> np.abs(yf[2]), np.rad2deg(np.angle(yf[2])) # 周期为 64 取样点的余弦波的振幅和相位
(0.25000000000000011, 44.999999999999993)
>>> np.abs(yf[3]), np.rad2deg(np.angle(yf[3])) # 周期为 42.66 取样点的余弦波的振幅和相位
(0.39999999999999991, -60.000000000000085)
```

这里 np.angle()计算复数的辐角，它得到的是弧度，通过 np.rad2deg()将弧度变换为角度。在这个例子中我们产生了 3 个频率、振幅和相位各不相同的余弦波：

- 周期为 128 个取样点的余弦波的相位为 0，振幅为 0.3。
- 周期为 64 个取样点的余弦波的相位为 45 度(π/4)，振幅为 0.5。
- 周期为 42.66(128/3.0)个取样点的余弦波的相位为-60(-π/3)度，振幅为 0.8。

对照 yf[1]、yf[2]、yf[3]的复数振幅和辐角，读者应该对 FFT 结果中的每个数值有了很清晰的理解。

FFT 的运算效率

FFT 的运算效率由 FFT 长度 N 的质因子决定，N 能被分解得越小则运算速度越快。例如，当 N 为素数时，FFT 的运算效率达到最低。下面的程序比较 4096 点 FFT 和 4093 点 FFT 运算的时间，由于 4096 是 2 的整数次幂，而 4093 是一个素数，因此它们的运算时间相差非常大：

```
>>> timeit.timeit("np.fft.fft(x)",
... "import numpy as np;x=np.random.random(4096)", number=100)
0.01888219968655136
>>> timeit.timeit("np.fft.fft(x)",
... "import numpy as np;x=np.random.random(4093)", number=100)
6.6385001708412261
```

15.1.2 合成时域信号

在 15.1.1 节的演示中，通过 ifft()可以将频域信号转换回时域信号，这种转换是精确的。下面的程序完成类似的频域信号转时域信号的计算。不过可以由用户选择一部分频域信号转换为时域信号，这样转换的结果和原始的时域信号会有误差，通过观察可以发现使用的频率信息越多，此误差越小。通过此程序可以直观地观察到多个余弦波的叠加是如何逐步逼近任意的时域信号的，图 15-1 显示了使用 FFT 计算的三角波频谱。



fft_example_triangle.py
使用 FFT 计算三角波的频谱

```
import numpy as np
import pylab as pl

# 产生 size 点取样的三角波，周期为 1
def triangle_wave(size): ❶
    x = np.arange(0, 1, 1.0/size)
    y = np.where(x<0.5, x, 0)
    y = np.where(x>=0.5, 1-x, y)
    return x, y

# 取 FFT 计算结果 freqs 中的前 n 项进行合成，返回合成结果，计算 loops 个周期的波形
def fft_combine(freqs, n, loops=1): ❷
    length = len(freqs) * loops
    data = np.zeros(length)
    index = loops * np.arange(0, length, 1.0) / length * (2 * np.pi)
    for k, p in enumerate(freqs[:n]):
        if k != 0: p *= 2 # 除去直流成分之外，其余的系数都乘以 2
        data += np.real(p) * np.cos(k*index) # 余弦成分的系数为实数部分
        data -= np.imag(p) * np.sin(k*index) # 正弦成分的系数为负的虚数部分
    return index, data

fft_size = 256

# 计算三角波及其 FFT 结果
x, y = triangle_wave(fft_size)
fy = np.fft.fft(y) / fft_size

# 绘制三角波的 FFT 的前 20 项的振幅，由于不含下标为偶数的值均为 0， 因此取
# log 之后无穷小，无法绘图，用 np.clip 函数设置数组值的上下限，保证绘图正确
pl.figure()
pl.plot(np.clip(20*np.log10(np.abs(fy[:20])), -120, 120), "o")
pl.xlabel(u"频率窗口(frequency bin)")
pl.ylabel(u"幅值(dB)")
```

```
# 绘制原始的三角波和用正弦波逐级合成的结果，使用取样点作为 x 轴坐标
pl.figure()
pl.plot(y, label=u"原始三角波", linewidth=2)
for i in [0,1,3,5,7,9]:
    index, data = fft_combine(fy, i+1, 2)  # 计算两个周期的合成波形
    pl.plot(data, label = "N=%s" % i)
pl.legend()
pl.show()
```

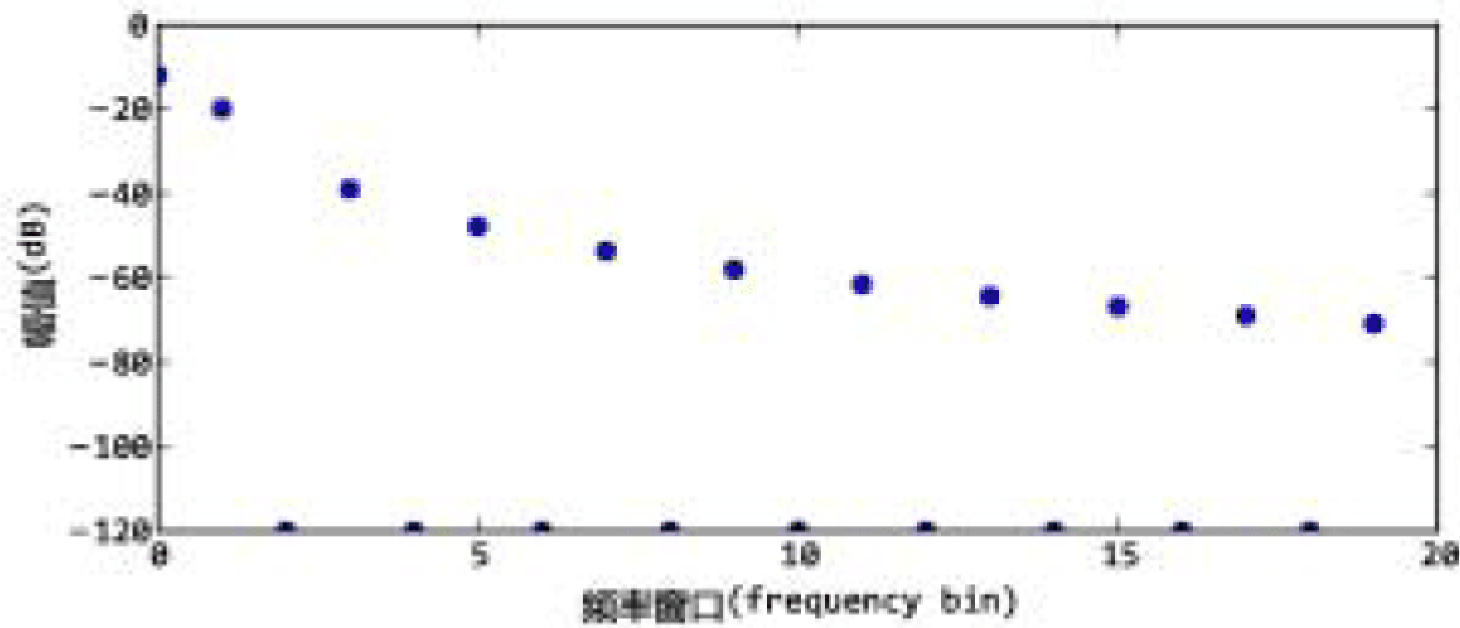


图 15-1　三角波的频谱

图 15-2 显示使用频谱中的部分频率重新合成的三角波(见文前彩插)。由此图可知，合成时域信号时，使用的频率越多，波形越接近原始的三角波。

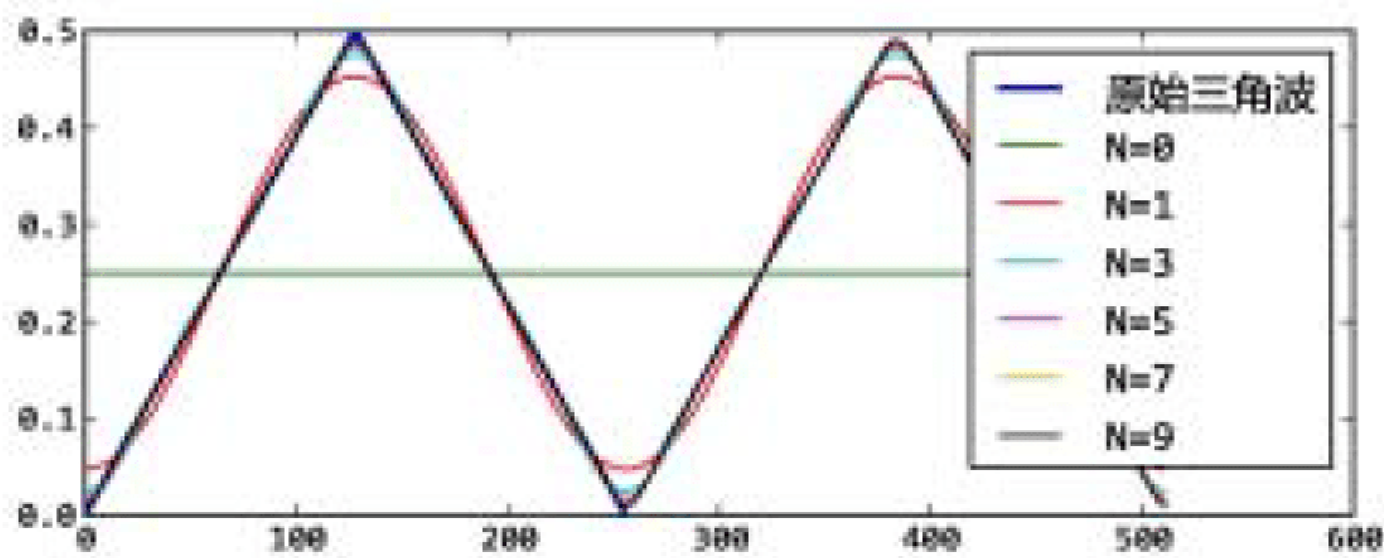


图 15-2　使用频谱中的部分频率重建的三角波

❶triangle_wave()产生一个周期的三角波，这里使用 np.where()计算区间函数的值。triangle()返回两个数组，分别表示 x 轴和 y 轴的值。注意后面的计算和绘图不使用 x 轴坐标，而是直接用取样次数作为 x 轴坐标。

❷fft_combine()使用 FFT 结果 freqs 中的前 n 个数据重新合成时域信号，loops 参数指定计算的周期数。

接下来再看看方波信号。由于方波的波形中存在跳变，因此用有限个正弦波合成的方波在跳变处会出现抖动现象，如图 15-3 所示，用正弦波合成的方波的收敛速度比三角波慢很多(见

文前彩插)。下面是计算方波波形的函数：



fft_example_rectangle.py
使用 FFT 计算方波的频谱

```
def square_wave(size):
    x = np.arange(0, 1, 1.0/size)
    y = np.where(x<0.5, 1.0, 0)
    return x, y
```

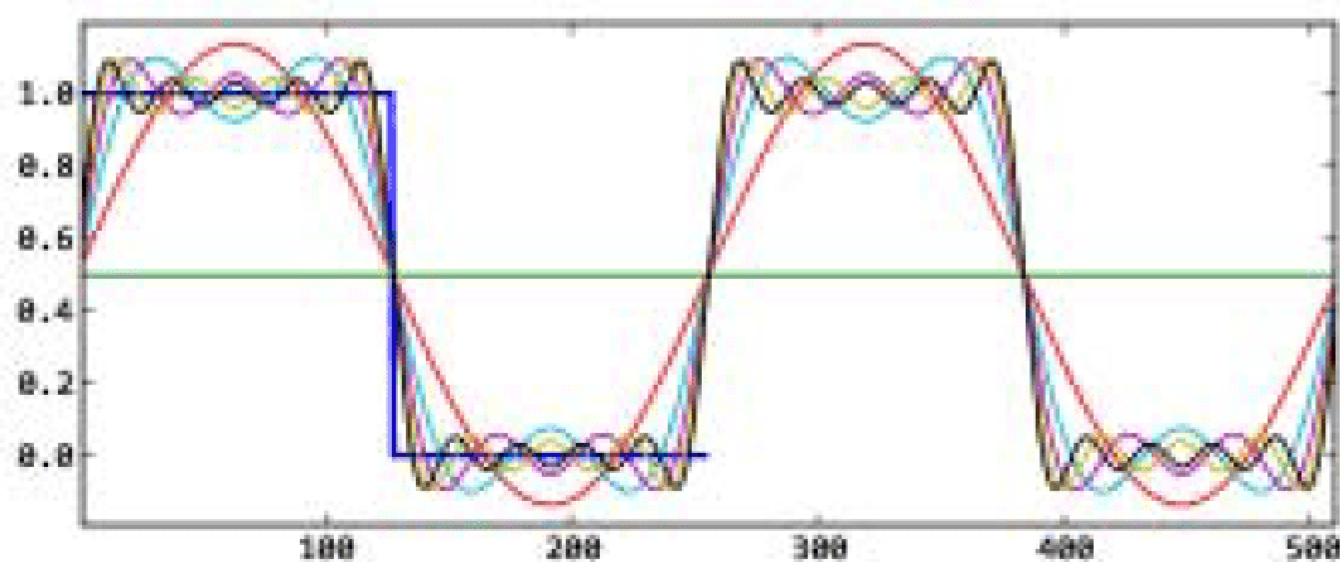


图 15-3　使用正弦波合成的方波在跳变处出现抖动

15.1.3　三角波 FFT 演示程序

最后让我们完成三角波的 FFT 演示程序，使用它可以交互式地观察各种三角波的频谱以及正弦合成的近似波形。制作界面是一件很费工夫的事情，幸好有 TraitsUI 库的帮忙，不到 200 行代码就可以制作出如图 15-4 所示的效果了。在此界面中，“波形宽度”等几个控件是一种特殊的数值编辑器，可以单击之后直接输入数值，也可以用鼠标左键左右拖动改变其数值。

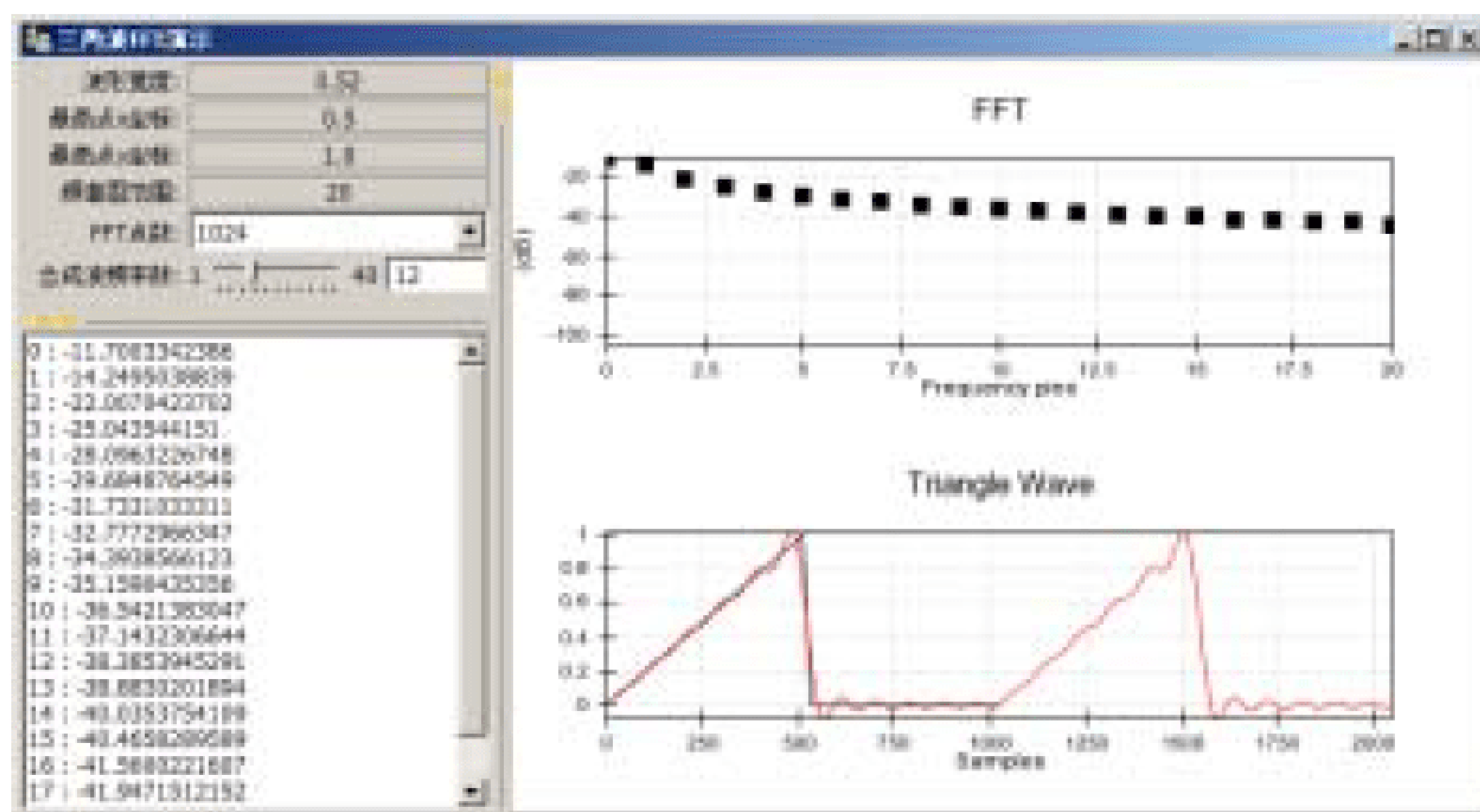


图 15-4　三角波频谱观察器界面



fft_triangle_GUI.py
三角波 FFT 演示程序

程序中已经给出了详细的注释，相信读者能够读懂并掌握这类程序的写法。

15.2 观察信号的频谱

将时域信号通过 FFT 转换为频域信号之后，将其各个频率分量的幅值绘制成图，便能很直观地观察信号的频谱。下面的程序可完成这一任务：



spectrum_full_period.py
整数个周期的正弦波频谱

```
import numpy as np
import pylab as pl

sampling_rate, fft_size = 8000, 512 ❶
t = np.arange(0, 1.0, 1.0/sampling_rate) ❷
x = np.sin(2*np.pi*156.25*t)  + 2*np.sin(2*np.pi*234.375*t) ❸
xs = x[:fft_size]
xf = np.fft.rfft(xs)/fft_size ❹
freqs = np.linspace(0, sampling_rate/2, fft_size/2+1) ❺
xfp = 20*np.log10(np.clip(np.abs(xf), 1e-20, 1e100)) ❻
pl.figure(figsize=(8,4))
pl.subplot(211)
pl.plot(t[:fft_size], xs)
pl.xlabel(u"时间(秒)")
pl.subplot(212)
pl.plot(freqs, xfp)
pl.xlabel(u"频率(Hz)")
pl.subplots_adjust(hspace=0.4)
pl.show()
```

❶首先定义了两个常数——sampling_rate 和 fft_size，分别表示数字信号的取样频率和 FFT 的长度。❷然后调用 np.arange()产生 1 秒钟的取样时间，t 中的每个数值直接表示取样点的时间，因此其间隔为取样周期 1/sampline_rate。

❸用取样时间数组 t 可以很方便地计算出波形数据，这里计算的是两个正弦波的叠加，一个频率是 156.25 Hz，一个是 234.375 Hz。为什么选择这两个奇怪的频率呢？因为这两个频率的正弦波在 512 个取样点中正好有整数个周期。只有整数个周期的波形的 FFT 结果能精确地反映

其频率。

N 点 FFT 能精确计算的频率

假设取样频率为 f_s，取波形中的 N 个数据进行 FFT 变换。那么当这 N 个数据包含整数个周期的波形时，FFT 计算的结果是精确的。于是能精确计算的波形的周期是 $n \cdot f_s / N$ 。对于 8k Hz 的取样频率、512 点的 FFT 来说，8000/512.0 = 15.625 Hz，即只能精确表示 15.625 Hz 的整数倍频率。156.25 Hz 和 234.375 Hz 正好是其 10 倍和 15 倍。

❹这里我们使用 rfft()对从波形数据 x 中截取的 fft_size 个取样点进行 FFT 计算，得到的结果不包括共轭部分。根据 FFT 计算公式，为了正确显示波形能量，还需要将结果除以 FFT 的长度 fft_size。

对于长度为 N 的 FFT 运算，rfft()返回 N/2+1 个复数，分别表示从 0 Hz 到(sampling_rate/2) Hz 的 N/2+1 点频率的成分。❺因此可以通过 linspace()计算出 rfft()的返回值中每个数值对应的真正频率。

❻最后计算每个频率分量的幅值，并将其转换为以 dB 度量的值。为了防止 0 幅值造成 log10()无法计算，我们调用 np.clip()对 xf 的幅值进行上下限处理。

剩下的程序将时域波形和频域波形绘制成如图 15-5 所示的图表。

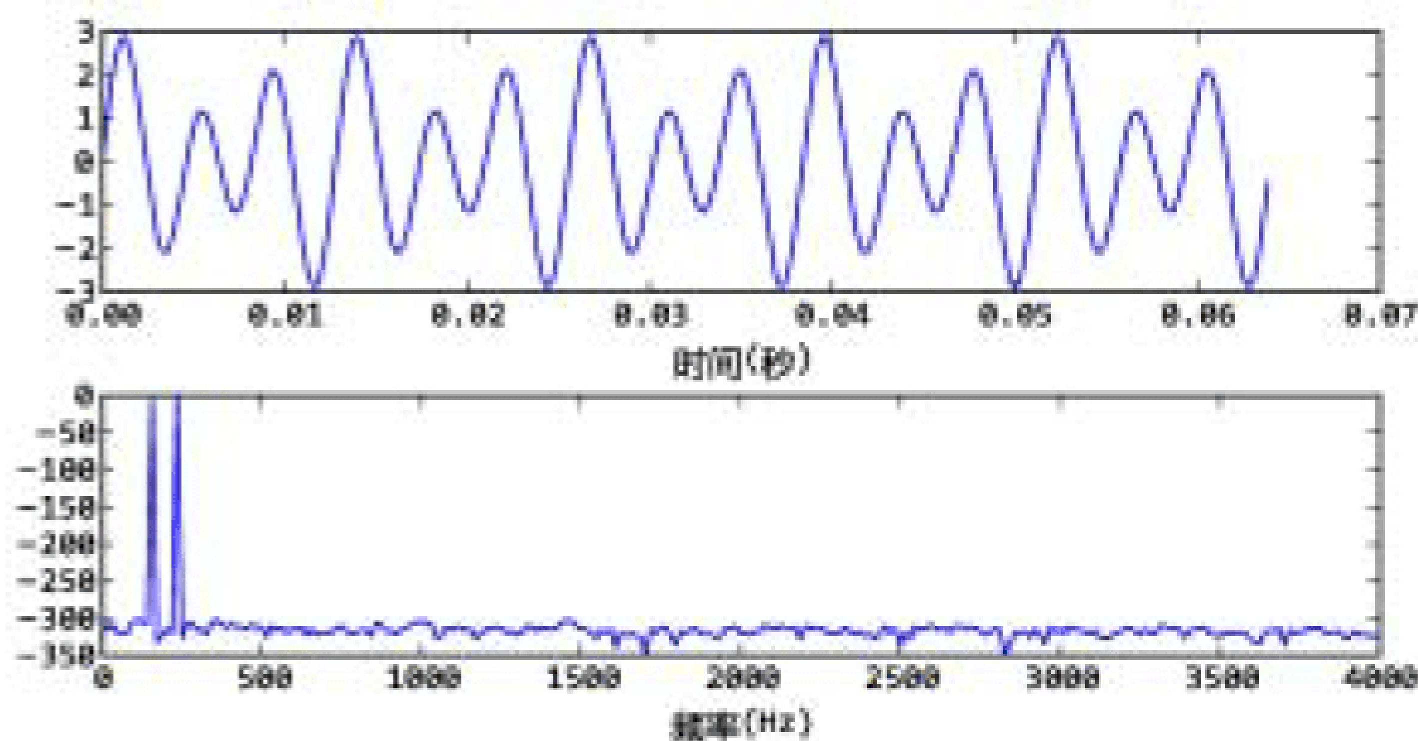


图 15-5　156.25 Hz 和 234.375 Hz 的波形(上)和频谱(下)

如果放大频谱中的两个峰值，就可以看到其值分别为：

```
>>> xfp[10]
-6.0205999132796251
>>> xfp[15]
-9.6432746655328714e-16
```

即 156.25 Hz 的幅值大小为-6 dB，而 234.375 Hz 的幅值大小为 0 dB。

下面我们看看非整数个周期的波形的频谱，将波形计算公式修改为：

```
x = np.sin(2*np.pi*200*t)  + 2*np.sin(2*np.pi*300*t)
```

得到的结果如图 15-6 所示。

spectrum_part_period.py
非整数周期的正弦波的频谱

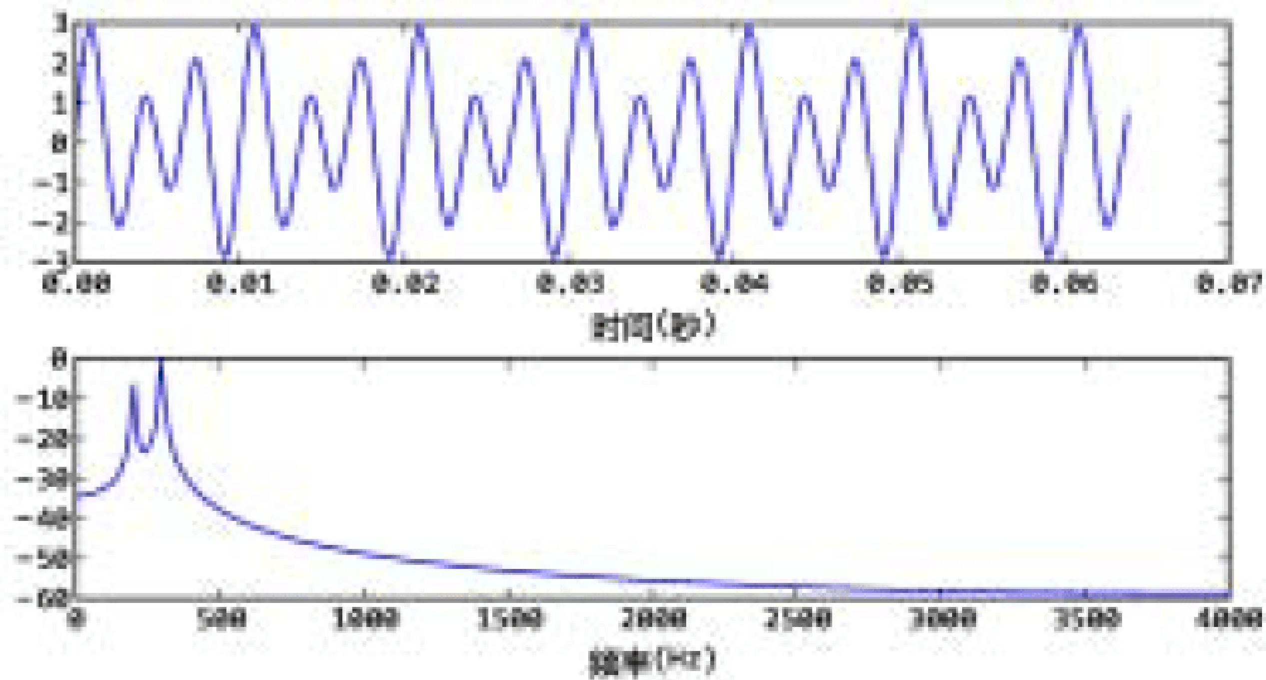


图 15-6　非完整周期(200 Hz 和 300 Hz)的正弦波经过 FFT 变换之后出现频谱泄漏

这次得到的频谱不再是两个完美的峰值，而是两个峰值频率周围的频率都有能量，这显然和两个正弦波的叠加波形的频谱有区别。本来应该属于 200 Hz 和 300 Hz 的能量分散到了周围的频率中，这个现象被称为频谱泄漏。出现频谱泄漏的原因在于 fft_size 个取样点无法放下整数个 200 Hz 和 300 Hz 的波形。

频谱泄漏的解释

我们只能在有限的时间段中对信号进行测量，无法知道测量范围之外的信号。因此只能对测量范围之外的信号进行假设。而傅立叶变换的假设很简单：测量范围之外的信号是所测量到的信号的重复。

现在考虑 512 点 FFT，从信号中取出的 512 个数据就是 FFT 的测量范围，它计算的是这 512 个数据一直重复的波形的频谱。显然如果 512 个数据包含整数个周期，那么得到的结果就是原始信号的频谱，而如果不是整数个周期，得到的频谱就是如图 15-7 的波形的频谱，图中的波形是用 8k Hz 的取样频率采集的 512 点的 50 Hz 的正弦波不断重复的结果。

spectrum_50 HzRepeat.py
计算 50 Hz 正弦波的 512 点取样的重复波形

```
>>> t = np.arange(0, 1.0, 1.0/8000)
>>> x = np.sin(2*np.pi*50*t)[:512]
```

```
>>> pl.plot(np.hstack([x,x,x]))
>>> pl.show()
```

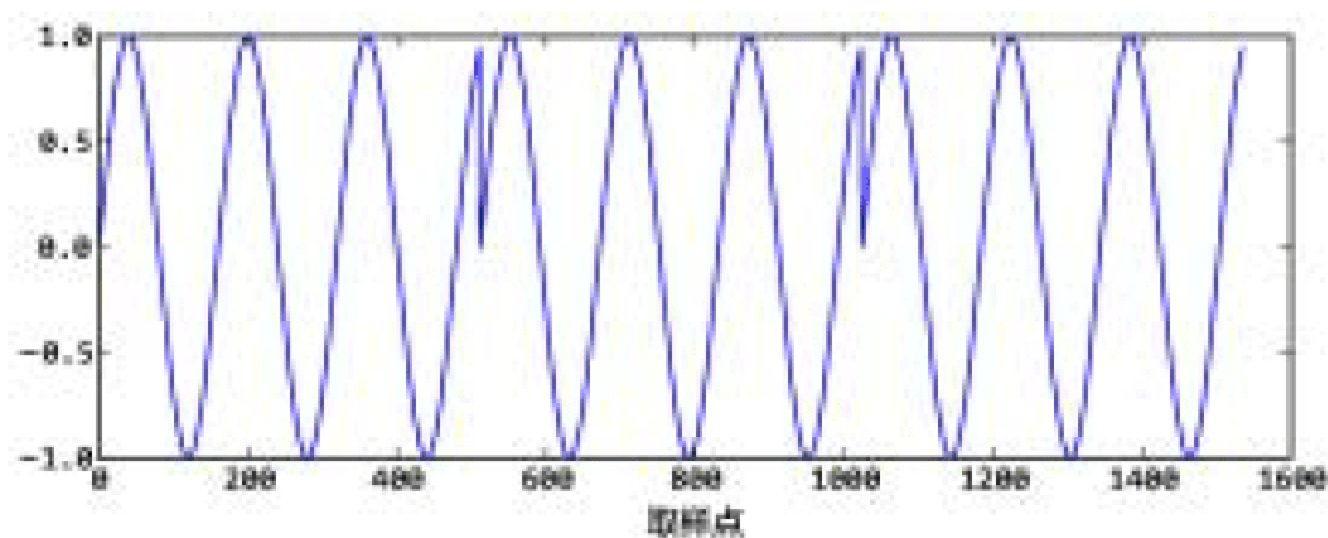


图 15-7　50 Hz 正弦波的 512 点 FFT 所计算的频谱的实际波形

由于波形的前后不是连续的，存在跳变，而跳变处有着非常广泛的频谱，因此 FFT 结果中将出现频谱泄漏。

15.2.1　窗函数

为了减少 FFT 所截取的数据段前后的跳变，可以让数据先乘以一个窗函数，使得其前后数据能平滑过渡。例如，常用的 Hann 窗函数的定义如下：

$$w(n) = 0.5(1 - \cos(\frac{2\pi n}{N-1}))$$

其中 N 为窗函数的点数，图 15-8 是一个 512 点 Hann 窗的曲线。


spectrum_hann_window.py
绘制 512 点 Hann 窗函数

```
>>> import pylab as pl
>>> import scipy.signal as signal
>>> pl.figure(figsize=(8,3))
>>> pl.plot(signal.hann(512))
```

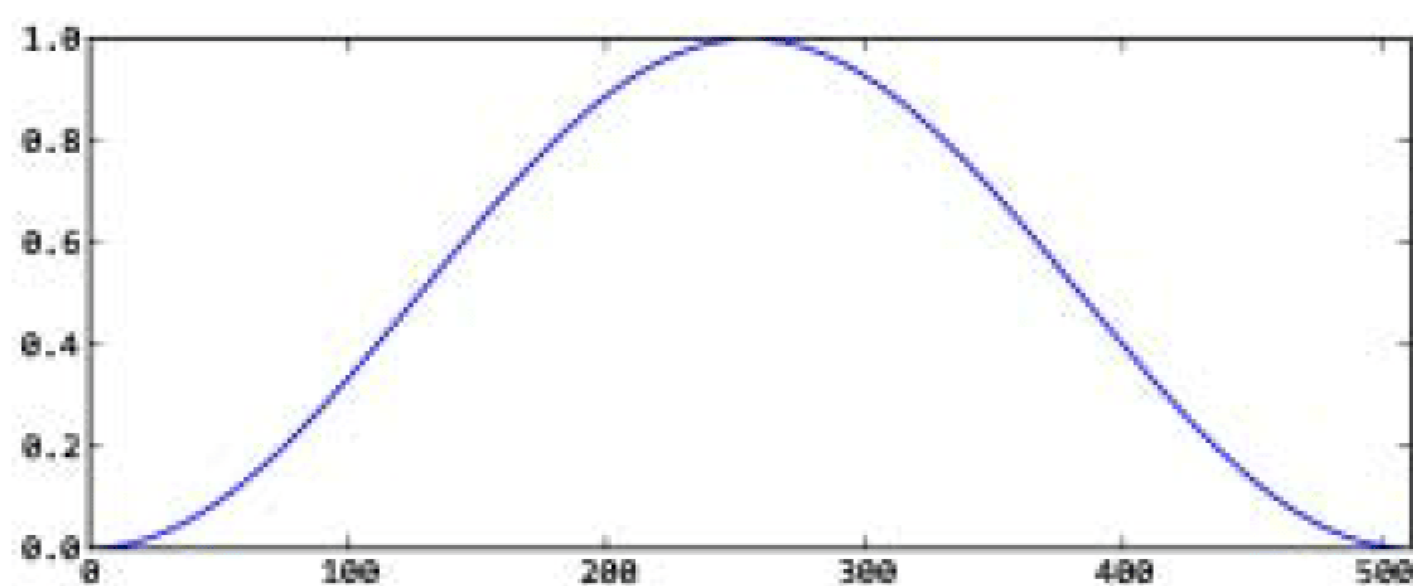


图 15-8　Hann 窗函数

窗函数都在 scipy.signal 库中定义，它们的第一个参数为点数 N。可以看出，Hann 窗函数是完全对称的，也就是说第 0 点和第 511 点的值完全相同，都为 0。如果将这样的窗函数与信号数据相乘，然而进行重复，结果中会出现前后两个连续的 0，这会对 FFT 变换之后的信号频谱有一定的影响。

为了解决连续 0 值的问题，Hann 窗函数提供了 sym 参数，如果它的值为 0，将产生一个 N+1 点的 Hann 窗函数，并舍去最末的数值，这样得到的窗函数适合于周期信号：

```
>>> signal.hann(8)
array([ 0.        ,  0.1882551 ,  0.61126047,  0.95048443,  0.95048443,
        0.61126047,  0.1882551 ,  0.        ])
>>> signal.hann(8, sym=0)
array([ 0.        ,  0.14644661,  0.5       ,  0.85355339,  1.        ,
        0.85355339,  0.5       ,  0.14644661])
```

50 Hz 正弦波与窗函数相乘之后的周期重复波形如图 15-9 所示。

spectrum_50 HzRepeat_hann.py
计算 50 Hz 正弦波与窗函数的乘积

```
>>> t = np.arange(0, 1.0, 1.0/8000)
>>> x = np.sin(2*np.pi*50*t)[:512] * signal.hann(512, sym=0)
>>> pl.plot(np.hstack([x,x,x]))
>>> pl.show()
```

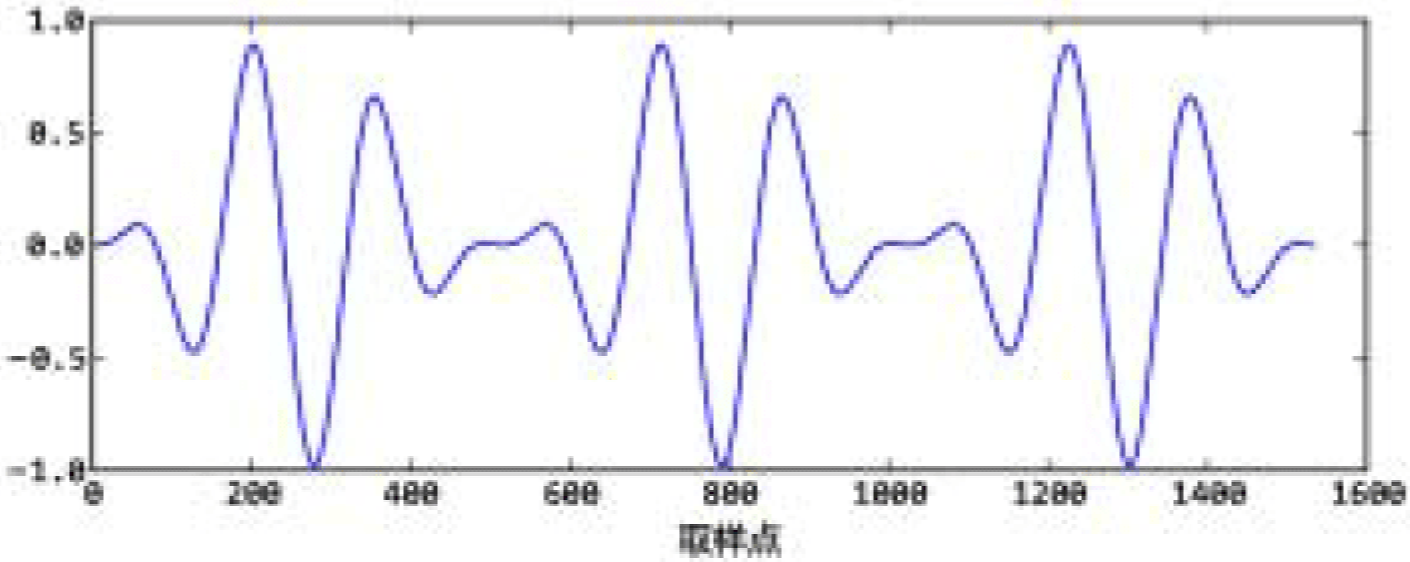


图 15-9　加 Hann 窗的 50 Hz 正弦波的 512 点 FFT 所计算的实际波形

回到前面的例子，将 200 Hz 和 300 Hz 的叠加波形与 Hann 窗相乘之后再计算其频谱，得到如图 15-10 的频谱图(见文前彩插)。

spectrum_hann_fft.py
使用 Hann 窗降低非整数周期的信号的频率泄漏

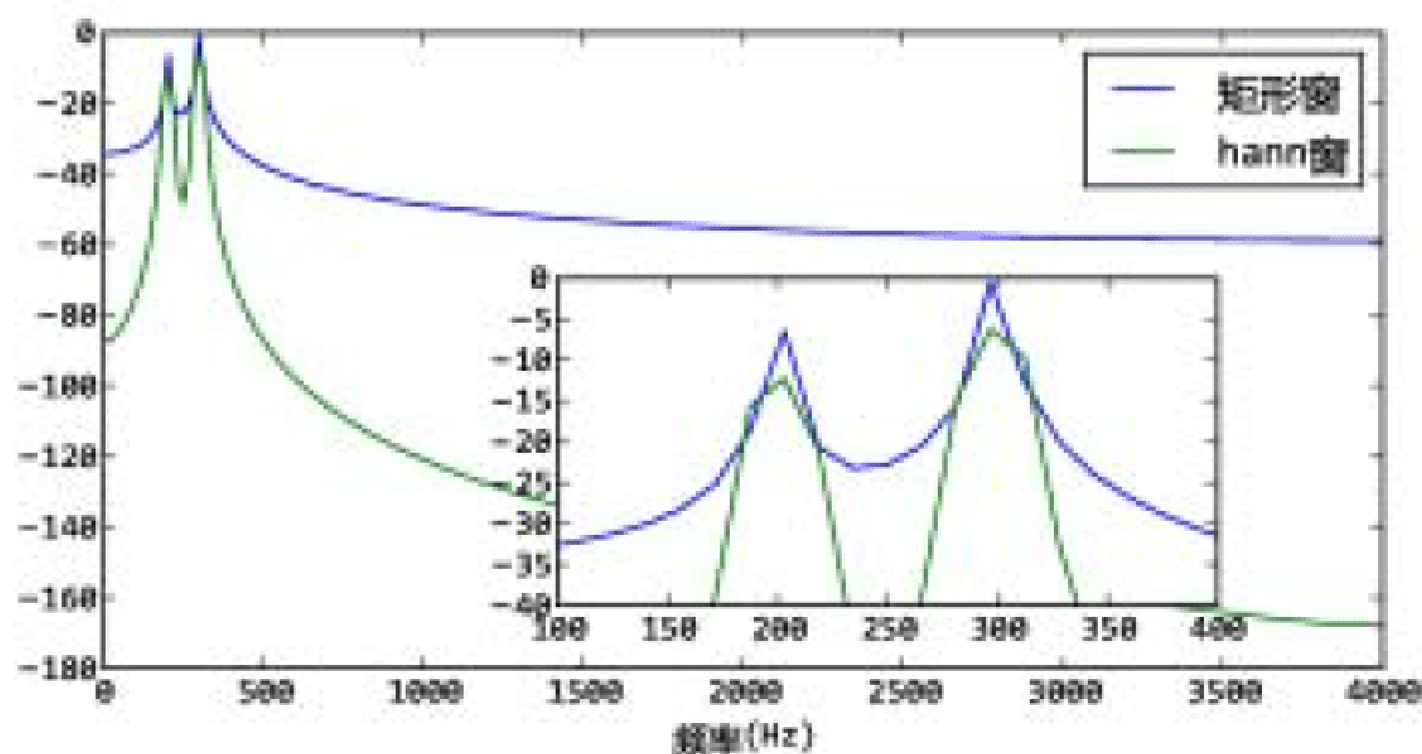


图 15-10　加 Hann 窗前后的频谱，Hann 窗能降低频谱泄漏

可以看到，与 Hann 窗相乘之后的信号的频谱能量更加集中于 200 Hz 和 300 Hz，但是其能量有所降低。这是因为 Hann 窗本身有一定的能量衰减：

```
>>> np.sum(signal.hann(512, sym=0))/512
0.5
```

因此，如果需要严格保持信号的能量，还需要在乘以 Hann 窗之后再把信号扩大一倍。

15.2.2　频谱平均

对于频谱特性不随时间变化的信号，例如引擎、压缩机等机器噪声，可以对其进行长时间的采样，然后分段进行 FFT 计算，最后对每个频率分量的幅值求平均值，便可以准确地测量信号的频谱。

下面的程序可完成这一计算：



spectrum_average_whitenoise.py
计算白噪声的平均频谱

```
def average_fft(x, fft_size):
    n = len(x) // fft_size * fft_size
    tmp = x[:n].reshape(-1, fft_size) ❶
    tmp *= signal.hann(fft_size, sym=0) ❷
    xf = np.abs(np.fft.rfft(tmp)/fft_size) ❸
    avgf = np.average(xf, axis=0) ❹
    return 20*np.log10(avgf)
```

average_fft(x, fft_size)对数组 x 进行 fft_size 点 FFT 运算，并返回以 dB 为度量的平均幅值。由于 x 的长度可能不是 fft_size 的整数倍，❶因此首先将其缩短为 fft_size 的整数倍，然后用

reshape()将其转换成一个二维数组 tmp。tmp 的第 1 轴的长度为 fft_size。

❷将 tmp 数组第 1 轴上的数据和 Hann 窗相乘。❸调用 rfft()对 tmp 数组的每行数据进行 FFT 计算，并求幅值。❹最后，用 average()对 xf 沿着第 0 轴进行平均，这样就得到每个频率分量的平均幅值。

图 15-11 是利用 average_fft()计算随机数序列频谱的例子。

```
>>> x = np.random.rand(100000) - 0.5
>>> xf = average_fft(x, 512)
>>> pl.plot(xf)
>>> pl.show()
```

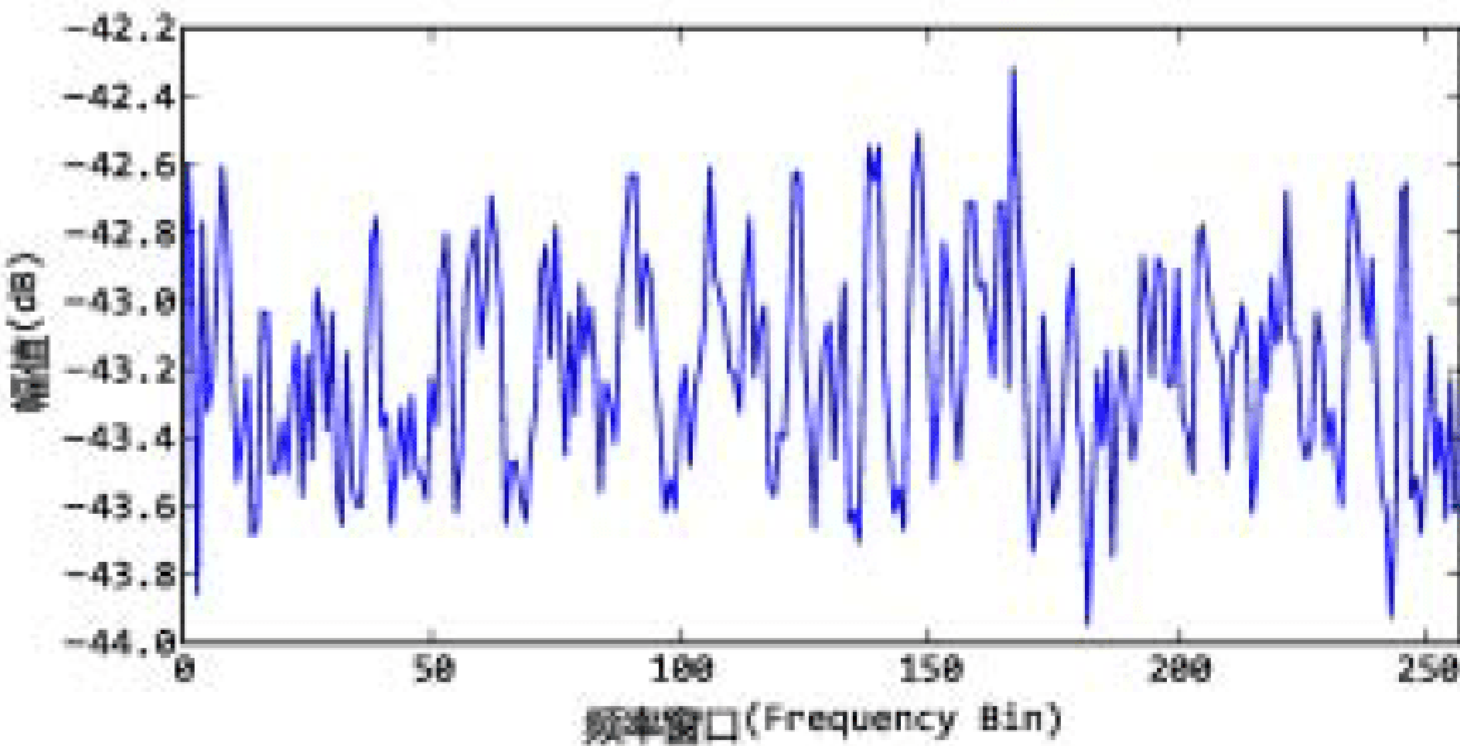


图 15-11　白噪声的频谱接近水平直线(注意 Y 轴的范围)

可以看到随机噪声的频谱接近一条水平的直线，也就是说每个频率窗口的能量都相同，这种噪声被称为白噪声。

如果我们让白噪声通过一个 IIR 低通滤波器，绘制其输出信号的平均频谱，就能够观察到 IIR 滤波器的频率响应特性。下面的程序利用 iirdesign()设计一个 8k Hz 取样的 1k Hz 的 Chebyshev I 型低通滤波器，iirdesign()需要用正规化的频率(取值范围为 0 到 1)，然后调用 filtfilt()对白噪声信号 x 进行低通滤波：



spectrum_average_lowpass.py
计算经过低通滤波器的白噪声的频谱

```
>>> b,a=signal.iirdesign(1000/4000.0, 1200/4000.0, 1, -40, 0, "cheby1")
>>> x = np.random.rand(100000) - 0.5
>>> y = signal.filtfilt(b, a, x)
```

如果用 average_fft()计算输出信号 y 的平均频谱，将得到如图 15-12 所示的频谱图。

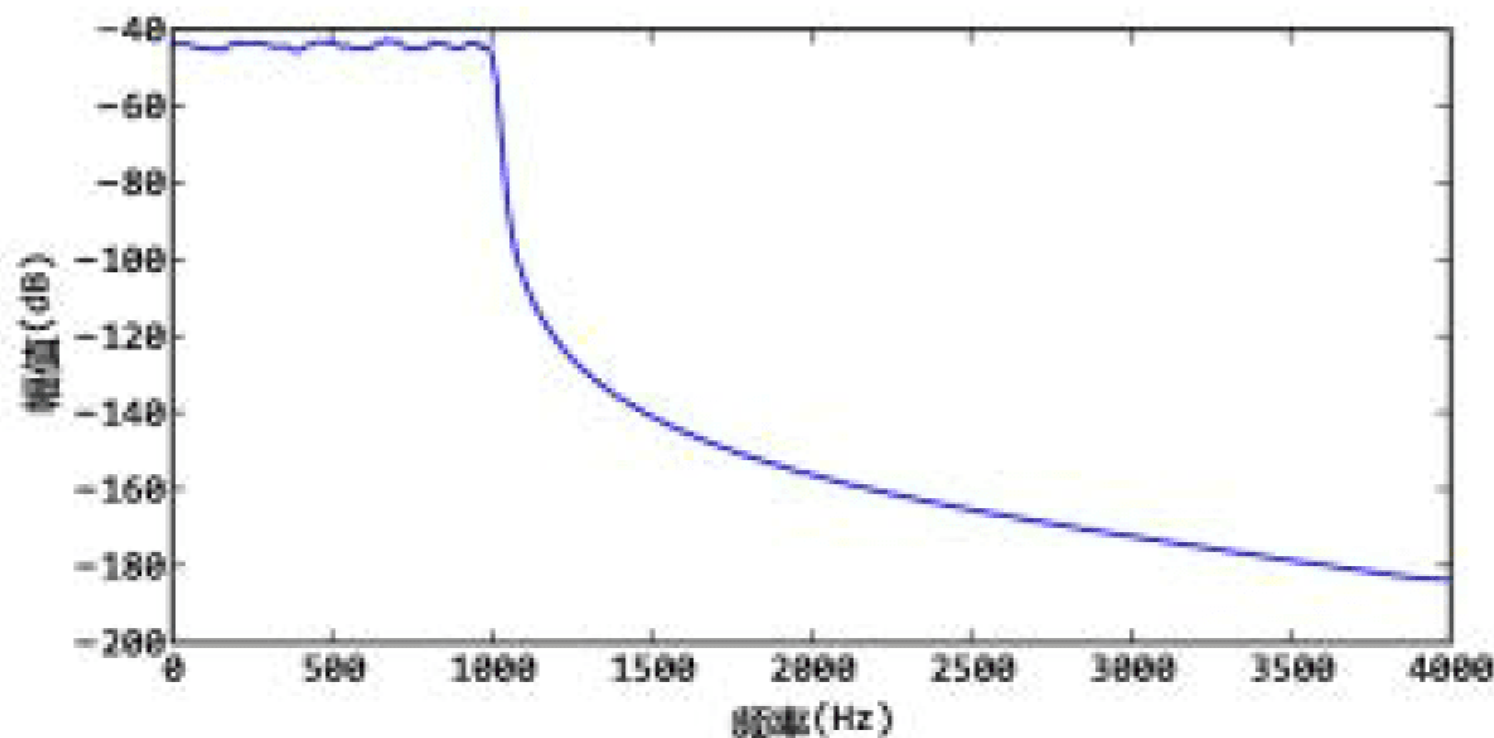


图 15-12　经过低通滤波器的白噪声的频谱

15.2.3　谱图

虽然使用 FFT 能够观察信号的频域特性，但是却完全丧失了信号在时间轴上的信息。因此前面介绍的观察信号频谱的方法只适合于频率特性不随时间变化的情况。当信号频率随时间变化时，为了既能观察信号频率又能观察其随时间的变化，可以使用短时距傅立叶变换(STFT[①])。

使用 STFT 算法得到的结果被称为谱图(Spectrogram)。谱图的横轴表示时间，纵轴表示频率，谱图上每点的值表示信号在此点的能量。STFT 算法其实很简单：对信号分段进行 FFT 处理，每一次处理的结果都是谱图中的一列。每段信号的长度越短，则时间轴上的精度越高，而频率轴上的精度却越低，反过来也是一样。时间轴和频率轴上的分辨率是一对不可调和的矛盾，根据傅立叶变换的不确定原理，我们不能指望同时获得频率和时间的高分辨率。

下面的程序用于绘制频率扫描波的谱图，效果如图 15-13 所示(见文前彩插)。通过此图可以很直观地观察到信号的频率随着时间而逐渐变高，并且是呈指数增长的。



spectrogram_sweep.py
绘制频率扫描波的谱图

```
import scipy.signal as signal
import pylab as pl
import numpy as np
from numpy.lib.stride_tricks import as_strided

sampling_rate = 8000.0
fft_size = 1024
step = fft_size/16 ❶
time = 2
```

① 英文全称为 Short-time Fourier transform。

```
t = np.arange(0, time, 1/sampling_rate)
sweep = signal.chirp(t, f0=100, t1 = time, f1=0.8*sampling_rate/2, method="logarithmic")
number = (len(sweep)-fft_size)/step ❷
data = as_strided(sweep, shape=(number, fft_size), strides=(step*8, 8)) ❸

window = signal.hann(fft_size) ❹
data = data * window

spectrogram = np.abs(np.fft.rfft(data, axis=1)) ❺
spectrogram /= fft_size/2
np.log10(spectrogram, spectrogram)
spectrogram *= 20.0

pl.figure(figsize=(8,4))
im = pl.imshow(spectrogram.T, origin = "lower",
    extent=[0, 2, 0, sampling_rate/2], aspect='auto') ❻
bar = pl.colorbar(im, fraction=0.05)
bar.set_label(u"能量(dB)")
pl.xlabel(u"时间(秒)")
pl.ylabel(u"频率(Hz)")
pl.show()
```

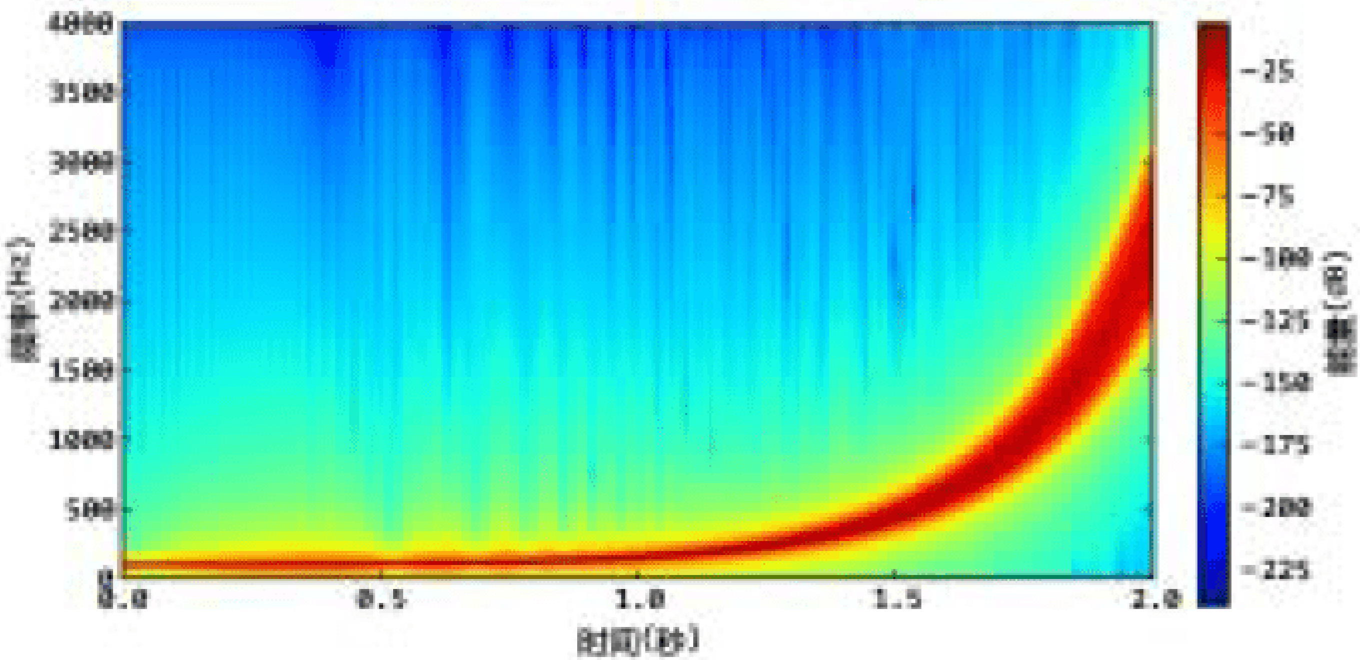


图 15-13　频率扫描波的谱图

❶为了使谱图在时间轴上的变化更平滑，我们设置连续两块数据的偏移量为 fft_size/16 个取样点。也就是说两块数据之间有 15/16 的部分是重复的。❷计算能将频率扫描波 sweep 分成的块数。❸使用 as_strided()将 sweep 的形状改为(number, fft_size)，并且设置 strides 的值。得到的结果 data 是一个二维数组，它的每一行对应原始信号的一个分块。

❹使用 Hann 窗对分块信号进行平滑处理，减少频谱泄漏。注意由于 data 中的每行之间有共用的数据，因此不能使用下面自更新的乘法：

```
data *= window
```

❺调用 rfft()对信号进行 FFT 计算，由于 axis 参数为 1，rfft()会对 data 中的每一行数据进行

FFT 运算。❻最后调用 imshow()绘制图像，并调用 colorbar()显示出颜色和值之间的关系。

实际上，pylab 模块中已经提供了绘制谱图的函数 specgram()，对于上面的频率扫描波，可以使用下面的语句绘制相同的谱图。其中，第一个参数是表示信号的数组，第二个参数是 FFT 的长度，第三个参数是信号的取样频率，noverlap 参数是连续两块数据之间重叠部分的长度。specgram()还有许多其他的关键字参数，请读者阅读函数文档以了解它的详细用法。

```
>>> import pylab as pl
>>> pl.specgram(sweep, fft_size, sampling_rate, noverlap = 1024-step)
```

下面是一个用 PyAudio、TraitsUI 及 Chaco 等制作的观察谱图的程序。它实时地从声卡读入声音数据，并绘制出声音信号的时间波形、频谱及谱图。由于本程序对计算机的配置要求较高，请读者在较快的机器上运行本程序。另外，为了计算效率，程序中没有使用窗函数和重叠处理。图 15-14 是程序的界面截图。



spectrogram_realtime.py
实时观察声音信号的谱图

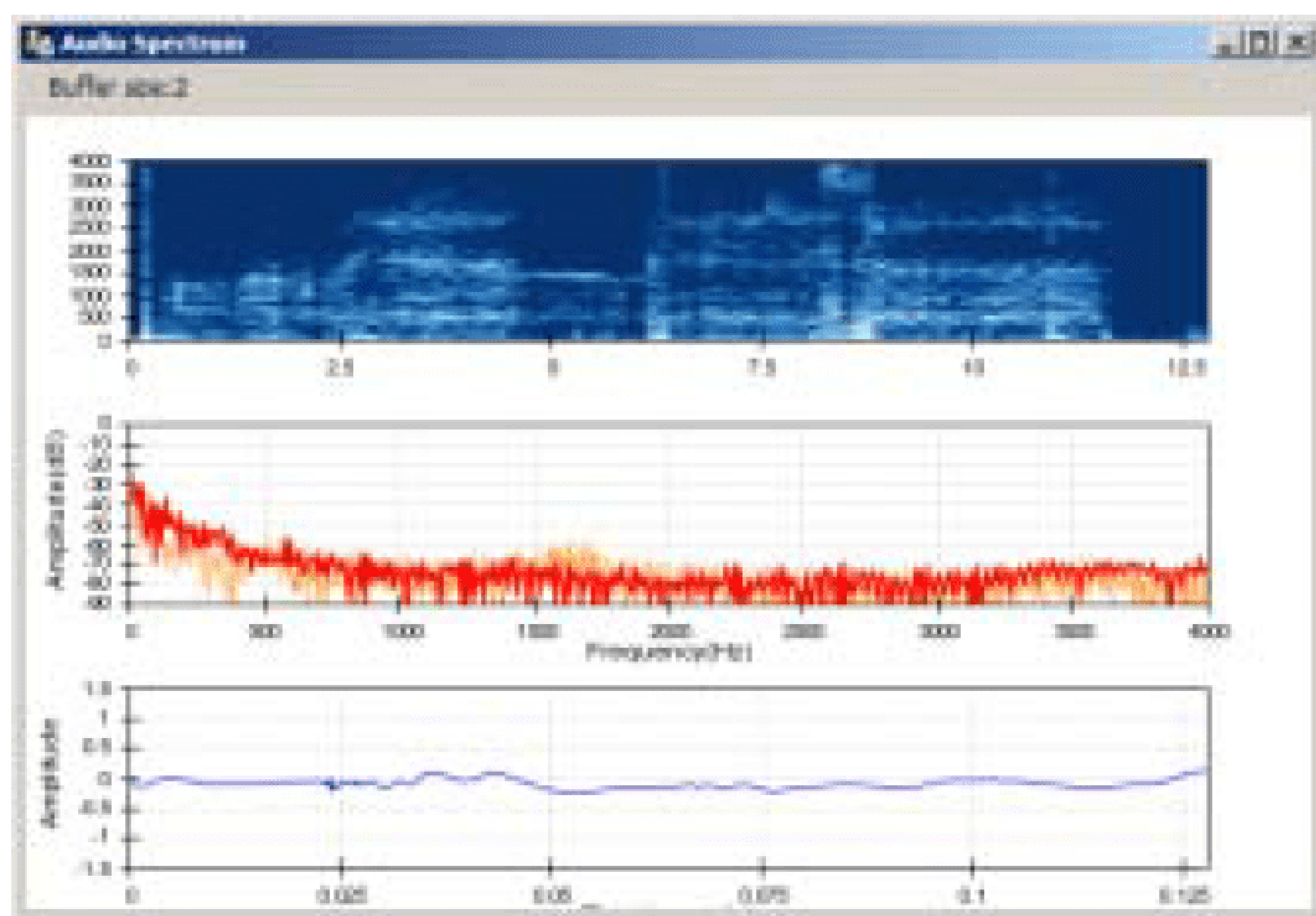


图 15-14　使用 TraitsUI 和 Chaco 制作的实时观察声音信号谱图的界面

使用 STFT 计算的谱图存在一个问题：它在时间轴和频率轴上的分辨率都是均一的，即谱图中的任意两块大小相同的区域所对应的时间范围和频率范围都完全相同。

然而，由于高频信号的变化速度很快，为了观察高频信号的变化，我们希望高频信号在时间轴上的分辨率高一些，也就是用更短的 FFT 计算频谱。而人类的听觉对于低频信号的频率变化更为敏感，因此对于低频信号我们希望在频率轴上的分辨率高一些，也就是用更长的 FFT 计算频率。

小波变换能够解决这个矛盾，在小波变换的谱图上，低频部分的频率分辨率高而时间分辨

率低，高频部分的时间分辨率高而频率分辨率低。感兴趣的读者可以阅读下面的教程：



http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html
关于小波变换和 STFT 的教程

15.3 卷积运算

我们知道，信号 x 经过系统 h 之后的输出 y 是 x 和 h 的卷积，虽然卷积的计算方法很简单，但是当 x 和 h 都很长的时候，卷积计算是非常耗费时间的。因此对于响应时间很长的系统 h，需要找到比直接计算卷积更快的方法。

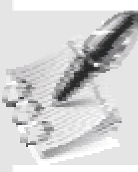
信号系统理论中有这样一个规律：时域的卷积等于频域的乘积，因此要计算时域的卷积，可以将时域信号转换为频域信号，进行乘积运算之后再将结果转换为时域信号，实现快速卷积。

15.3.1 快速卷积

由于 FFT 运算可以高效地将时域信号转换为频域信号，其运算的复杂度为$O(N \cdot \log N)$，因此三次 FFT 运算加一次乘积运算的总复杂度仍然为$O(N \cdot \log N)$级别，而卷积运算的复杂度为$O(N^2)$，显然通过 FFT 计算卷积要比直接计算快得多(这里假设需要卷积的两个信号的长度都为 N)。

但是有一个问题：FFT 运算假设计算的信号为周期信号，因此通过上述方法计算出的结果实际上是两个信号的循环卷积，而不是线性卷积。为了用 FFT 计算线性卷积，需要对信号进行补零扩展，使得其长度大于线性卷积结果的长度。

例如，如果要计算数组 a 和 b 的卷积，a 和 b 的长度都为 128，那么它们的卷积结果的长度为 $\text{len(a)} + \text{len(b)} - 1 = 255$。为了让 FFT 能够计算线性卷积，需要将 a 和 b 的长度都扩展到 256。下面的程序演示了这个计算过程：



spectrum_fft_convolve.py
使用 FFT 进行快速卷积运算

```
import numpy as np

def fft_convolve(a,b):
    n = len(a)+len(b)-1
    N = 2**(int(np.log2(n))+1) ❶
    A = np.fft.fft(a, N) ❷
    B = np.fft.fft(b, N)
    return np.fft.ifft(A*B)[:n] ❸
```

```
if __name__ == "__main__":
    a = np.random.rand(128)
    b = np.random.rand(128)
    c = np.convolve(a,b)
    print np.sum(np.abs(c - fft_convolve(a,b)))
```

程序输出直接卷积和 FFT 快速卷积的结果之间的误差，大约为 5e－12 左右。

在这段程序中，a、b 的长度为 128，卷积结果 c 的长度为 n=255。❶找到大于 n 的最小的 2 的整数次幂。❷fft()的第二个参数为 FFT 的长度，当输入数据的长度不够时，fft()自动对其进行补零。❸最后对两个频域信号的乘积调用 ifft()，得到时域信号的卷积。结果比实际的卷积结果多一个数，这个多出来的数值应该接近于 0，请读者自行验证。

由于直接计算卷积和使用 FFT 的快速卷积的复杂度级别不同，因此当卷积数据很长时，可以观察到明显的速度差别。下面的程序比较两种卷积算法的运算时间：

```
>>> import timeit
>>> setup="""import numpy as np
a=np.random.rand(10000)
b=np.random.rand(10000)
from spectrum_fft_convolve import fft_convolve"""
>>> timeit.timeit("np.convolve(a,b)",setup, number=10)
1.852900578146091
>>> timeit.timeit("fft_convolve(a,b)",setup, number=10)
0.19475575806416145
```

显然，计算两个很长的数组的卷积，FFT 快速卷积要比直接卷积快很多。但是对于较短的数组，直接卷积运算将更快一些。图 15-15 显示了直接卷积和快速卷积的平均计算时间和长度之间的关系(见文前彩插)。其中，Y 轴显示的是每个数据的平均计算时间。



Spectrum_fft_convolve_timeit.py
比较直接卷积和快速卷积的计算时间

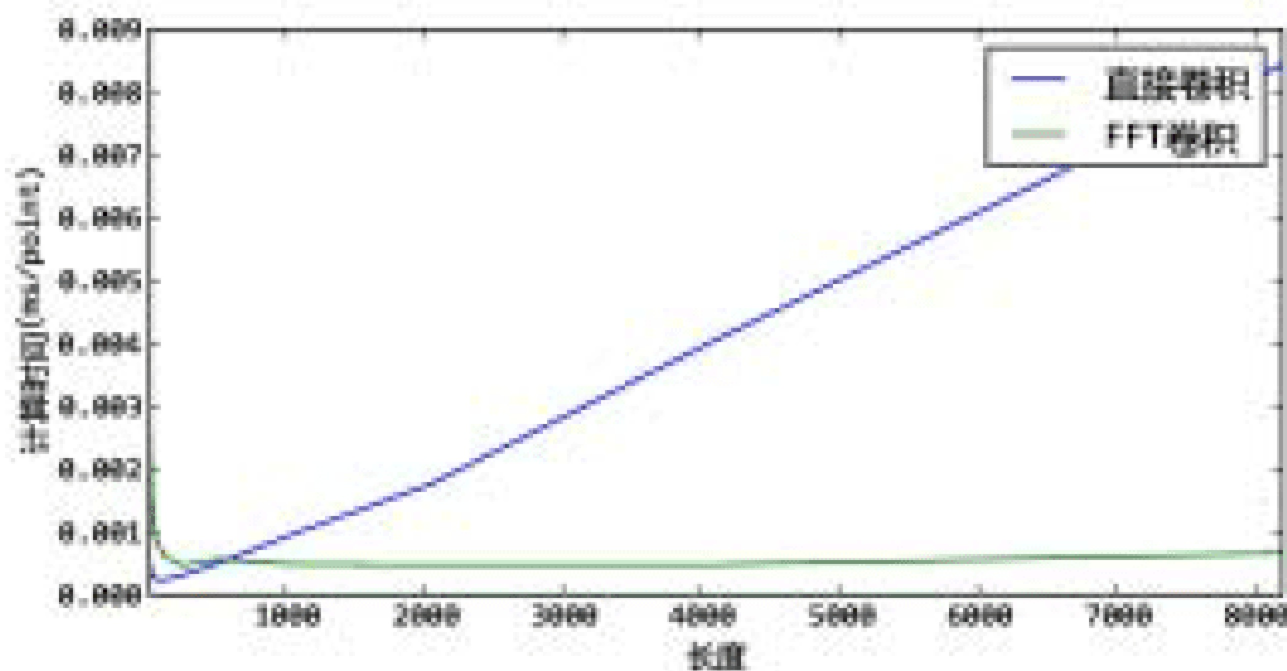


图 15-15　FFT 快速卷积和直接卷积的时间复杂度比较

由于图15-15中Y轴显示的是每个数据的计算时间，因此对于直接卷积它是线性的：$O(N^2)/N$。我们看到对于长度大于1024的计算，快速卷积显示出明显的优势。

由于FFT卷积很常用，因此scipy.signal库中提供了fftconvolve()，此函数采用FFT运算，并可计算多维数组的卷积。

15.3.2 分段运算

现在考虑对于输入信号x和系统响应h的卷积运算。通常输入信号是非常长的，例如要对某段录音进行滤波处理，假设取样频率为8k Hz，录音长度为1分钟，则数据的长度为480000。而如果需要对麦克风的输入信号进行连续的滤波处理，那么输入信号的长度可以看做无限长。系统响应h的长度通常都是固定的，例如它可能是某个房间的脉冲响应，或者是某种FIR滤波器的系数。

根据前面的介绍，为了有效地利用FFT计算卷积，我们希望它的两个输入长度相当，于是就需要对输入信号进行分段处理。卷积的分段运算被称为overlap-add运算。

overlap-add运算的计算方法如图15-16所示。

spectrum_overlap_add_plot.py
绘制overlap-add运算演示图

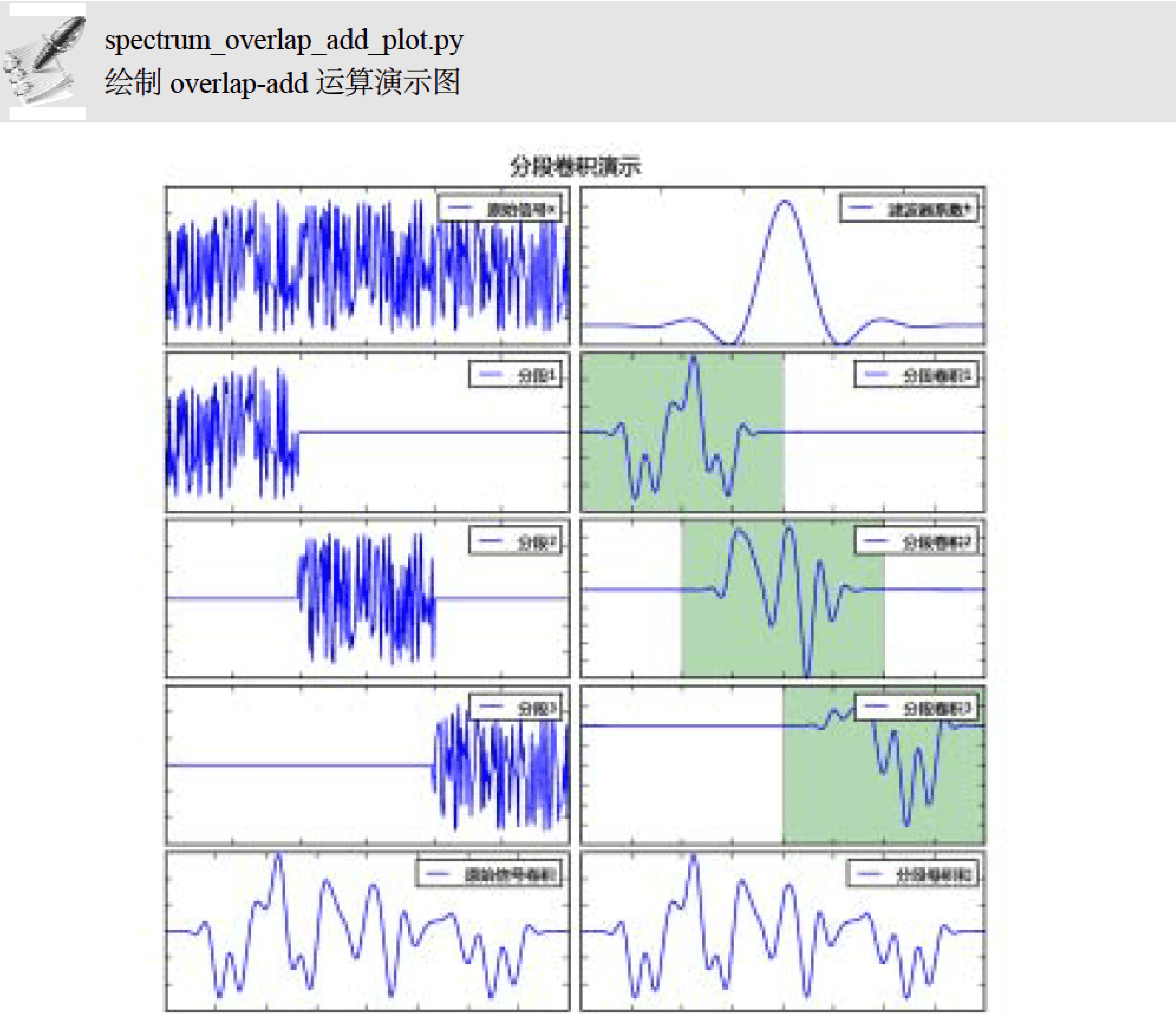


图15-16 使用overlap-add进行分段卷积的过程演示

图中原始信号 x 的长度为 300，将它分为三段，分别与滤波器系数 h 进行卷积计算，h 的长度为 101，因此每段输出 200 个数据，图中用绿色标出每段输出的 200 个数据。这 3 段数据按照时间顺序进行求和之后，得到的结果和原始信号的卷积是相同的。

因此将持续的输入信号 x 和滤波器 h 进行卷积的运算可以按照如下步骤进行，假设 h 的长度为 M：

(1) 建立一个缓存，大小为 N + M − 1，初始值为 0。

(2) 每次从 x 中读取 N 个数据，和 h 进行卷积，得到 N + M − 1 个数据。将这些数与缓存中的数据进行求和，并将结果保存到缓存中，最后输出缓存的前 N 个数据。

(3) 将缓存中的数据向左移动 N 个元素，也就是让缓存中的第 N 个元素成为第 0 个元素，左移完成之后将缓存中后面的 N 个元素全部设置为 0。

(4) 跳转到步骤(2)重复进行。

下面是实现这一算法的演示程序：



spectrum_overlap_add.py
分段卷积演示

```
import numpy as np
x = np.random.rand(1000)
h = np.random.rand(101)
y = np.convolve(x, h)

N = 50 # 分段大小
M = len(h) # 滤波器长度

output = []

#缓存初始化为0
buffer = np.zeros(M+N-1,dtype=np.float64)

for i in xrange(len(x)/N):
    #从输入信号中读取N个数据
    xslice = x[i*N:(i+1)*N]
    #计算卷积
    yslice = np.convolve(xslice, h)
    #将卷积的结果加入到缓存中
    buffer += yslice
    #输出缓存中的前N个数据，注意使用copy，否则输出的是buffer的一个视图
    output.append( buffer[:N].copy() ) ❶
    #缓存中的数据左移N个元素
    buffer[0:-N] = buffer[N:]
    #后面的补0
    buffer[-N:] = 0
```

```
#将输出的数据组合为数组
y2 = np.hstack(output)
#计算和直接卷积的结果之间的误差
print "error:", np.sum(np.abs( y2 - y[:len(x)] ) )
```

❶注意需要输出缓存中前N个数据的副本，否则输出的是数组的一个视图，当此后缓存buffer更新时，视图中的数据会一起更新。程序输出直接卷积和 overlap-add 分段卷积的误差：

```
error: 2.11408668349e-12
```

将 FFT 快速卷积和 overlap-add 相结合，可以制作出一些快速的实时数据滤波算法。但是由于 FFT 卷积对于两个长度相当的数组最为有效，因此在分段时也会有所限制。例如，如果滤波器的长度为 2048，那么理想的分段长度也为 2048；如果将分段长度设置得过低，反而会增加运算量。因此在实时性要求很强的系统中，只能采用直接卷积。

15.4 信号处理

有些信号处理算法很难对时域信号直接处理，而在频域进行计算则会很简单。频域信号处理的大致步骤如图 15-17 所示。

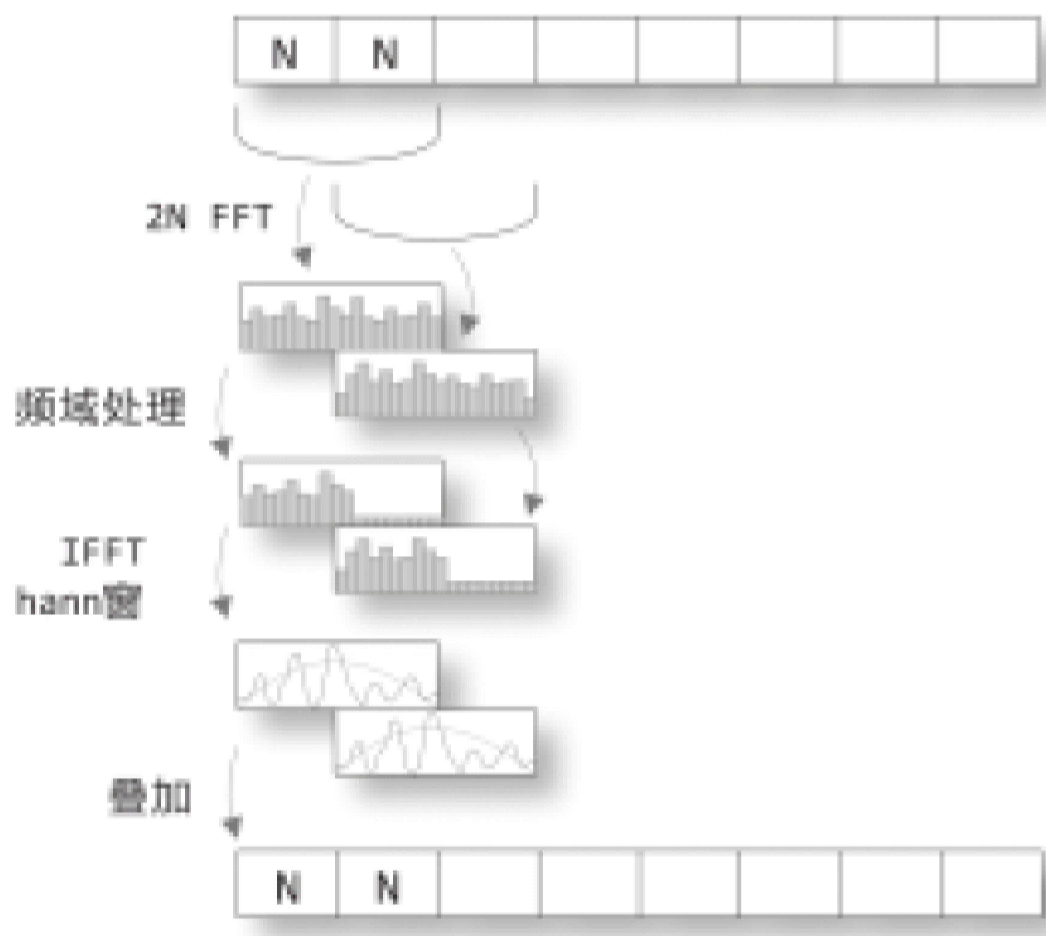


图 15-17 频域信号处理示意图

(1) 首先将输入信号以每 N 个取样为一组进行分块，N 为 2 的整数次幂。
(2) 以 1/2 的重叠率对 2*N 个取样信号进行 FFT 计算。
(3) 对 FFT 的计算结果进行频域信号处理。
(4) 用 IFFT 将处理后的频域信号转换为时域信号。

(5) 每块时域信号都与 2*N 点的 Hann 窗进行乘积运算。

(6) 将得到的时域信号以 1/2 的重叠率进行叠加，得到处理后的时域信号。

这种处理算法直接对信号的频谱进行修改，然后再通过 IFFT 转换为时域信号。由于是分块计算，转换后的信号之间没有很好的连续性，因此需要用 1/2 重叠率和 Hann 窗在两个信号块之间进行平滑过度。

15.4.1 基本框架

从算法流程可以看出，除了频域信号处理部分会根据应用的不同而改变之外，其余部分的运算都是不变的。因此可以编写如下用于频域处理的基类程序，实现处理中通用部分的计算，然后由其派生类实现不同的频域信号处理算法。



spectrum_freq_process.py
频域信号处理的基类程序

```
import numpy as np
import wave

class FrequencyProcess(object):
    def __init__(self, infile, outfile, fft_size): ❶
        self.fft_size = fft_size

        if type(infile) == str:
            f = wave.open(infile, "rb")
            nchannels, sampwidth, framerate, nframes, _, _ = f.getparams()
            if nchannels != 1:
                print "only support one channel wave file"
            self.framerate = framerate

            self.input = np.fromstring(f.readframes(nframes), dtype=np.short)
            f.close()
        else:
            self.input = infile
            self.framerate = 44100

        self.result = self.process()

        f = wave.open(outfile, "wb")
        f.setnchannels(1)
        f.setsampwidth(2)
        f.setframerate(self.framerate)
        f.writeframes(self.result.astype(np.short).tostring())
        f.close()
```

```
    def process_init(self):
        "频域信号处理的初始化"
        pass

    def process_block(self, block):
        "对频域信号块 block 进行处理，并返回"
        return block

    def process(self):
        self.process_init() ❷
        out = np.zeros(len(self.input))
        window = np.hanning(self.fft_size)

        start = 0
        while start + self.fft_size < len(self.input):
            block_time = self.input[start:start+self.fft_size]
            block_freq = np.fft.rfft(block_time)
            block_freq = self.process_block(block_freq)  ❸
            block_time = np.fft.irfft(block_freq)
            out[start:start+self.fft_size] += block_time * window # hanning 窗
            start += self.fft_size//2

        return out
```

❶在对象初始化方法__init__()中，首先从输入 WAVE 文件中读入声音数据，然后调用 process()对输入数据进行频域信号处理，最后将处理之后的数据写到输出 WAVE 文件中。为了方便演示后面将要介绍的各种频域处理算法，如果参数 infile 不是字符串，就直接把它当做数组使用，此时的取样频率固定为 44.1k Hz。

在 process()中，❷首先调用 process_init()进行频域处理算法的初始化，基类中的 process_init()不做任何事情，需要用其派生类覆盖此方法。然后以 1/2 重叠率，对输入数据进行 FFT 和 IFFT 运算。❸对于计算得到的每个频域数据块 block_freq，都调用 process_block()对其进行处理，process_block()也需要用派生类覆盖。

15.4.2 频域滤波器

对频域信号中的每个频率分量都乘以不同的系数，就能形成一个频域信号滤波器。它和时域滤波器相比，具有易于设计及计算速度快等特点。下面是频域滤波器的程序代码：



spectrum_freq_filter.py
频域滤波器

```
import numpy as np
from scipy.interpolate import UnivariateSpline
```

```
from spectrum_freq_process import FrequencyProcess

class FrequencyFilter(FrequencyProcess):
    def __init__(self, infile, outfile, fft_size, parameters): ❶
        self.parameters = parameters
        super(FrequencyFilter, self).__init__(infile, outfile, fft_size)

    def process_init(self):
        self.freqs = np.arange(0, self.fft_size//2+1, 1.0)
        freq_points = []
        freq_gains = []
        for freq, gain in self.parameters:
            # 实际频率转换为frequency bin
            freq_points.append(float(freq) / self.framerate * self.fft_size)
            freq_gains.append( 10**(gain/20.0) ) # dB 转换为乘积系数
        gain_func = UnivariateSpline(freq_points, freq_gains, k=1, s=0) ❷
        self.gain = gain_func(self.freqs)

    def process_block(self, block):
        return block * self.gain ❸

if __name__ == "__main__":
    from scipy import signal
    t = np.arange(0, 10, 1/44100.0)
    sweep = signal.chirp(t, f0=100, t1 = 10, f1=4000) * 10000
    FrequencyFilter(sweep, "chrip_filter.wav", 1024,
        [(0,0),(200, 0),(400, -20),(1000, -20),
        (1500, 0),(3000, 0),(3500, -20),(30000, -20)])
    FrequencyFilter("voice.wav", "voice_filter.wav", 1024, filter_settings)
```

❶初始化方法__init__()中的 parameters 参数是一组(频率[Hz],增益[dB])的列表。❷用一阶 UnivariateSpline 对象对 parameters 参数进行线性插值，得出每个频率窗口对应的增益系数。由于在滤波计算时，系数是固定不变的，因此系数的计算在 process_init()中进行。❸而 process_block()则只是简单地返回频率信号与其对应的系数的乘积。

为了测试程序的效果，我们对频率扫描波数据进行比较复杂的滤波。图 15-18 是程序输出的 WAVE 文件的波形。

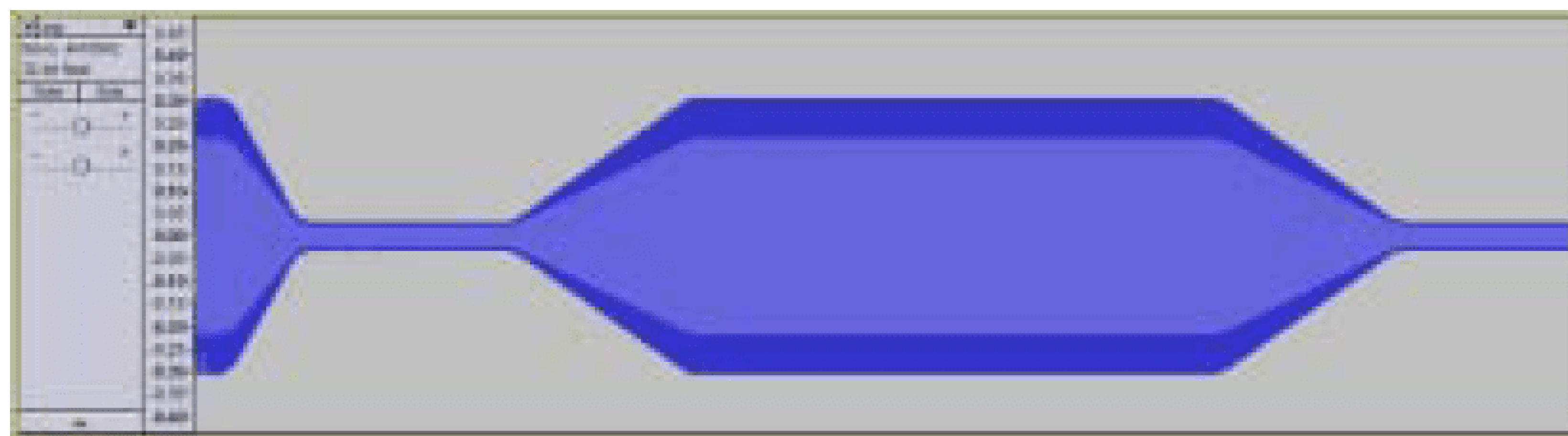


图 15-18　频域滤波器对频率扫描波的滤波结果

由于我们先将以 dB 为比例度量的增益参数转换为乘积系数，再对其进行线性插值，因此波形的振幅都是线性变化的。读者可以修改程序，先对增益参数进行线性插值，再转换为乘积系数，看看波形振幅有何变化。

15.4.3 频率变调处理

对频域信号进行压缩或拉伸，将改变对应的声音信号的音调，即可以把男声变为女声或者把女声变为男声。下面是声音变调程序：



spectrum_freq_scale.py
声音变调

```
import numpy as np
from spectrum_freq_process import FrequencyProcess
from scipy.interpolate import UnivariateSpline

class FrequencyScale(FrequencyProcess):
    def __init__(self, infile, outfile, fft_size, scale):
        self.scale = scale
        super(FrequencyScale, self).__init__(infile, outfile, fft_size)

    def process_init(self):
        self.freqs = np.arange(0, self.fft_size//2+1, 1.0)

    def process_block(self, block):
        new_freqs = self.freqs * self.scale
        freqs_re = UnivariateSpline(new_freqs, np.real(block), k=3, s=0) ❶
        freqs_im = UnivariateSpline(new_freqs, np.imag(block), k=3, s=0)

        block = freqs_re(self.freqs) + 1j*freqs_im(self.freqs) ❷
        block[int(np.max(new_freqs)):] = 0  ❸
        return block
if __name__ == "__main__":
    FrequencyScale("voice.wav", "voice_fscale.wav", 1024, 0.8)
```

❶使用两个 UnivariateSpline 对象分别对频域信号 block 的实数部分和虚数部分进行插值。❷注意这里使用的是变调之后的频率窗口作为 X 轴的数据，然后再使用原来的频率窗口对两个样条曲线进行求值，计算变调之后每个频率窗口对应的值。

例如，当 scale 为 2.0 时，原来的 100 Hz 的频率成分，就变成了 200 Hz 的频率成分，相当于沿着频率轴拉伸了频率信号。❸当 scale<1.0 时，相当于对频率信号进行压缩处理，因此高频部分会超出样条曲线的取值范围，变成外推计算，因此需要将这部分外推计算的结果修改为 0。

此程序处理之后的声音效果并不是很好，更完美的变调算法需要计算声音波形的自相似情况，找到合适的波形分割点进行分块处理。

15.4.4 用谱图差减法降噪

在使用麦克风录制声音信号时，由于受设备的电气噪声或环境噪声的影响，录制的声音信号通常会带有一定的背景噪声，例如常见的“沙—沙—”声。通常这种背景噪声都具有不变性，即它的频谱特性不会随着时间的变化而改变。这时可以使用谱图差减法去除背景噪声，具体算法如下：

(1) 录制一段纯背景噪声。

(2) 对背景噪声计算频谱能量的平均值。

(3) 对声音信号进行分段 FFT，对于每个频率 f 都按照如下公式计算其增益系数，其中 P_y 和 P_n 分别为声音信号和噪声的能量。

$$gain(f) = \min[(P_y(f) - \alpha P_n(f)) / P_y(f), \beta)]$$

(4) 使用 $gain(f)$对输入信号的频谱进行乘积计算，得到输出信号的频谱。

下面是实现此算法的程序：



spectrum_spectral_subtract.py
使用谱图差减法去除背景噪声

```
import numpy as np
from spectrum_freq_process import FrequencyProcess

def average_spectrum(data, fft_size):
    "以 1/2 的覆盖率计算 data 信号的每个频率的平均能量"
    p = np.zeros(fft_size//2+1)
    start = 0
    n = 0
    while start + fft_size < len(data):
        tmp = np.abs(np.fft.rfft(data[start:start+fft_size]))
        p += tmp*tmp
        n += 1
        start+=fft_size//2
    p /= n
    return p

class SpectrumSubtract(FrequencyProcess):
    def __init__(self, infile, outfile, fft_size, noise_len, a, b):
        self.noise_len = noise_len
        self.a = a
        self.b = b
        super(SpectrumSubtract, self).__init__(infile, outfile, fft_size)
```

```
    def process_init(self):
        self.noise = average_spectrum(self.input[:self.noise_len], self.fft_size)
        self.avg_gain = np.zeros(self.fft_size//2+1)
        # 频率平滑用的卷积窗口
        self.moving_window = np.hanning(9)
        self.moving_window /= np.sum(self.moving_window)
        self.moving_size = len(self.moving_window)//2

    def process_block(self, block):
        block_power = np.abs(block) ** 2 #信号 block 的能量

        # 由谱相减得到的每个频率窗口的 gain
        gain = (block_power - self.a*self.noise)/block_power
        np.clip(gain, self.b, 1e20, gain)

        # 对 gain 在频率上进行平滑处理
        gain = np.convolve(gain, self.moving_window)
        gain = gain[self.moving_size:self.moving_size+self.fft_size//2+1]

        # 对 gain 在时间上进行平滑处理
        self.avg_gain *= 0.8
        gain *= 0.2
        self.avg_gain += gain

        # 处理块
        block *= self.avg_gain
        return block
if __name__ == "__main__":
    SpectrumSubtract("voice.wav", "voice_ss.wav", 2048, 138000, 1.2, 0.05)
```

程序中假设输入的声音信号的最前面是一段无信号的背景噪声，通过 noise_len 参数指定这段背景噪声的长度。除了实现上述算法之外，程序还对每个块计算所得的 $gain(f)$进行频率之间的平滑处理，以及时间上(前后块之间)的平滑处理。这样能够防止声音信号产生过度的变化，降低谱图差减法带来的音乐噪声(musical noise)。两种平滑处理的参数都是固定的：频率平滑窗口为 H 点 Hann 窗，而时间平滑参数为 0.8。读者可以将其修改为传入的参数，测试它们对输出声音的影响。

15.5 Hilbert 变换

Hilbert 变换能在振幅保持不变的情况下将输入信号的相角偏移 90°，简单地说就是能将正弦波转换为余弦波，下面的程序可验证这一特性：



spectrum_hilbert_sin.py
正弦波的 Hilbert 变换

```
from scipy import fftpack
import numpy as np
import matplotlib.pyplot as pl

# 产生 1024 点 4 个周期的正弦波
t = np.linspace(0, 8*np.pi, 1024, endpoint=False)
x = np.sin(t)

# 进行 Hilbert 变换
y = fftpack.hilbert(x)
pl.plot(x, label=u"原始波形")
pl.plot(y, label=u"Hilbert 转换后的波形")
pl.legend()
pl.show()
```

程序的输出如图 15-19 所示，hilbert()将正弦波变换成了余弦波(见文前彩插)。

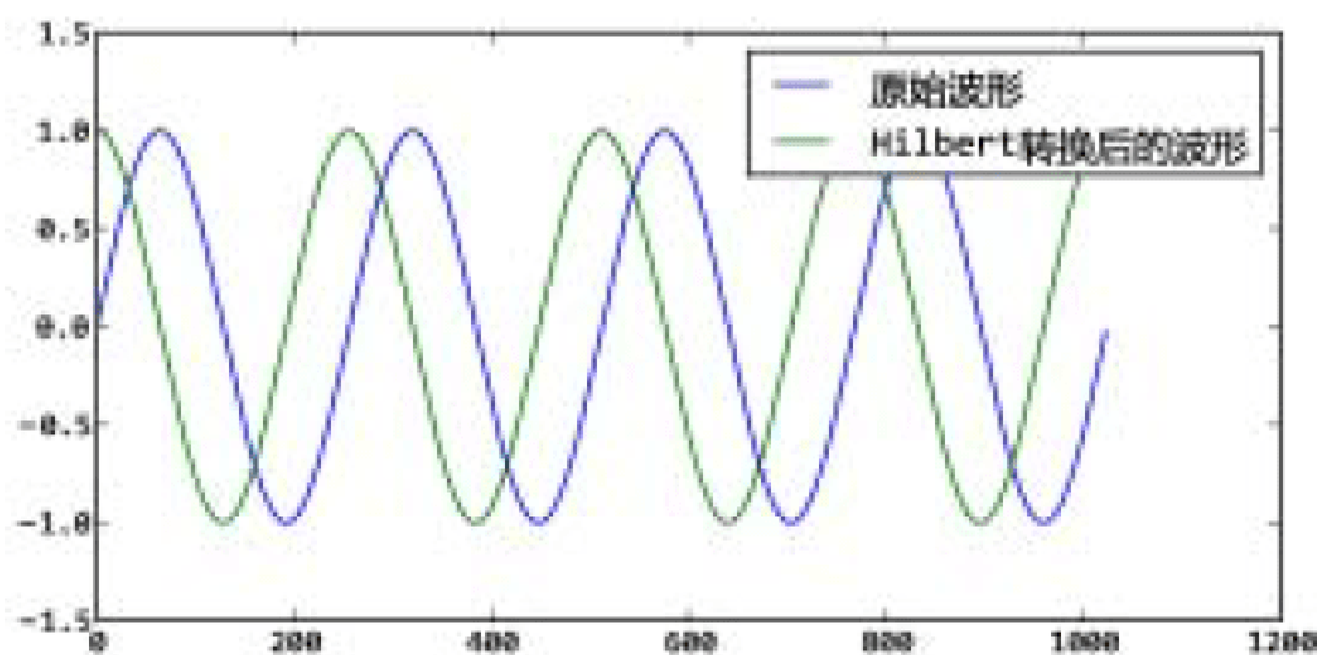


图 15-19　Hilbert 变换将正弦波变换为余弦波

Hilbert 正变换的相角偏移符号

本书将相角偏移+90°称为 Hilbert 正变换。有的文献或书籍则正好将定义倒转过来：将偏移+90°称为 Hilbert 负变换，而将偏移-90°称为 Hilbert 正变换。

相角偏移 90°相当于复数平面上的点与虚数单位 1j 相乘，因此 Hilbert 变换的频率响应可以用如下公式给出：

$$H(\omega) = j \cdot \mathrm{sgn}(\omega)$$

其中，ω 为圆频率，sgn 函数为符号函数，即：

$$\operatorname{sgn}(\omega)=\begin{cases}1, & \omega>0\\ 0, & \omega=0\\ -1, & \omega<0\end{cases}$$

我们可以将其频率响应理解为：

- 直流分量为 0
- 正频率成分偏移+90°
- 负频率成分偏移-90°

对于实数信号来说，正负频率成分共轭，因此对实数信号进行 Hilbert 变换之后仍然是实数信号。下面的程序验证 Hilbert 变换的频率响应：



spectrum_hilbert_freq.py
使用 FFT 验证 Hilbert 变换的频率响应

```
>>> x = np.random.rand(16)
>>> y = fftpack.hilbert(x)
>>> X = np.fft.fft(x)
>>> Y = np.fft.fft(y)
>>> np.imag(Y/X)
array([ 0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        0., -1., -1., -1., -1., -1., -1., -1.])
```

对信号进行 N 点 FFT 变换之后：

- 下标为 0 的频率分量表示直流分量
- 下标为 N/2 的频率分量为取样频率/2 的频率分量
- 1 到 N/2－1 为正频率分量
- N/2+1 到 N 为负频率分量

对照 Y/X 的虚数部分，不难看出它符合 Hilbert 频率响应。如果用 np.real(Y/X)观察实数部分，可以看到它们全部接近于 0。

Hilbert 变换可以用作包络检波，具体算法如下所示：

$$envelope=\sqrt{H(x)^2+x^2}$$

其中，x 为原始载波波形，H(x)为 x 经 Hilbert 变换之后的波形，envelope 为信号 x 的包络。其原理很容易理解：假设 x 为正弦波，那么 H(x)为余弦波，根据公式

$$\sin^2(t)+\cos^2(t)=1$$

可知 envelope 恒等于 1，即为 sin(t)信号的包络。下面的程序验证这一算法，程序的输出如图 15-20 所示(见文前彩插)。



spectrum_hilbert_envelop.py
用 Hilbert 变换进行包络检波

```
import numpy as np
import pylab as pl
from scipy import fftpack

t = np.arange(0, 0.3, 1/20000.0)
x = np.sin(2*np.pi*1000*t) * (np.sin(2*np.pi*10*t) + np.sin(2*np.pi*7*t) + 3.0)
hx = fftpack.hilbert(x)

pl.plot(x, label=u"载波信号")
pl.plot(np.sqrt(x**2 + hx**2), "r", linewidth=2, label=u"检出的包络信号")
pl.legend()
pl.show()
```

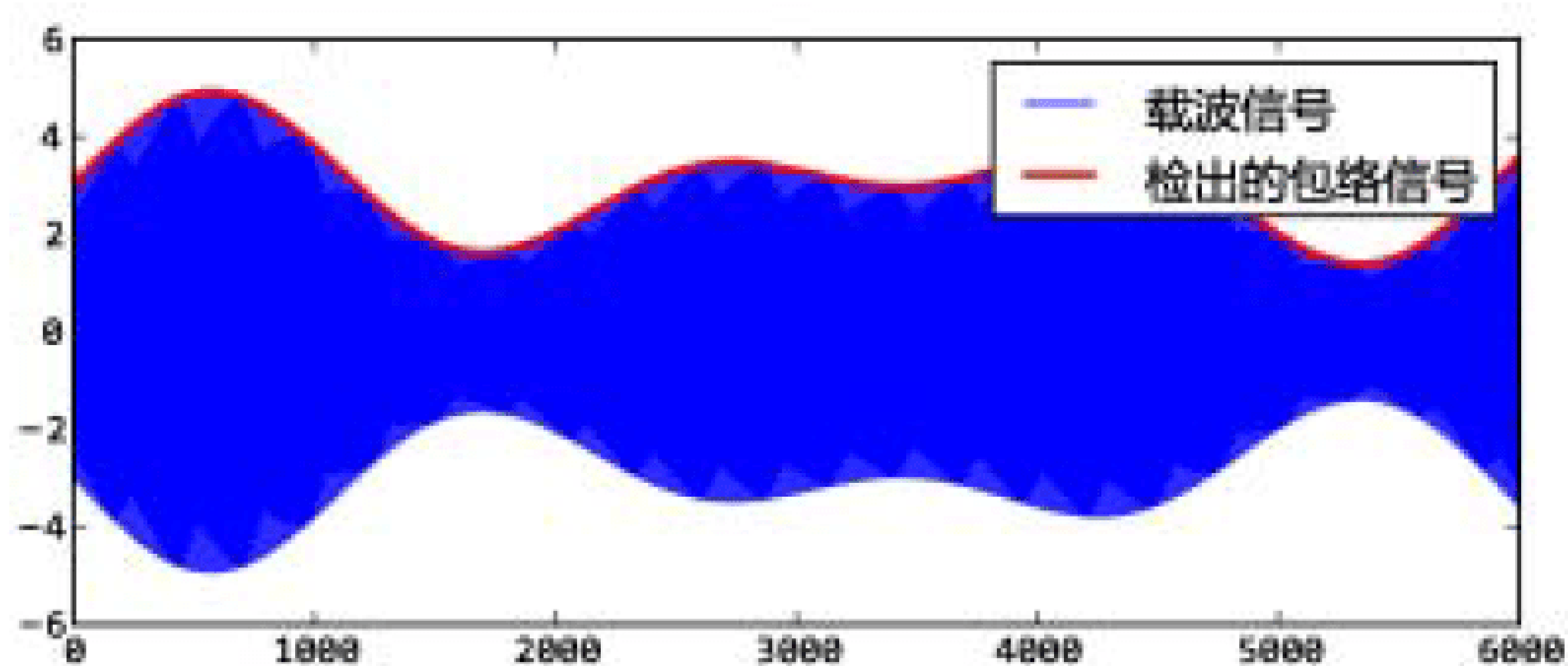


图 15-20　使用 Hilbert 变换对载波信号进行包络检波

前面介绍过可以使用频率扫描波测量滤波器的频率响应，我们可以使用这个算法计算出扫描波的包络。



spectrum_hilbert_sweep_envelop.py
用 Hilbert 变换计算频率扫描波的包络

得到的包络波形如图 15-21 所示(见文前彩插)。

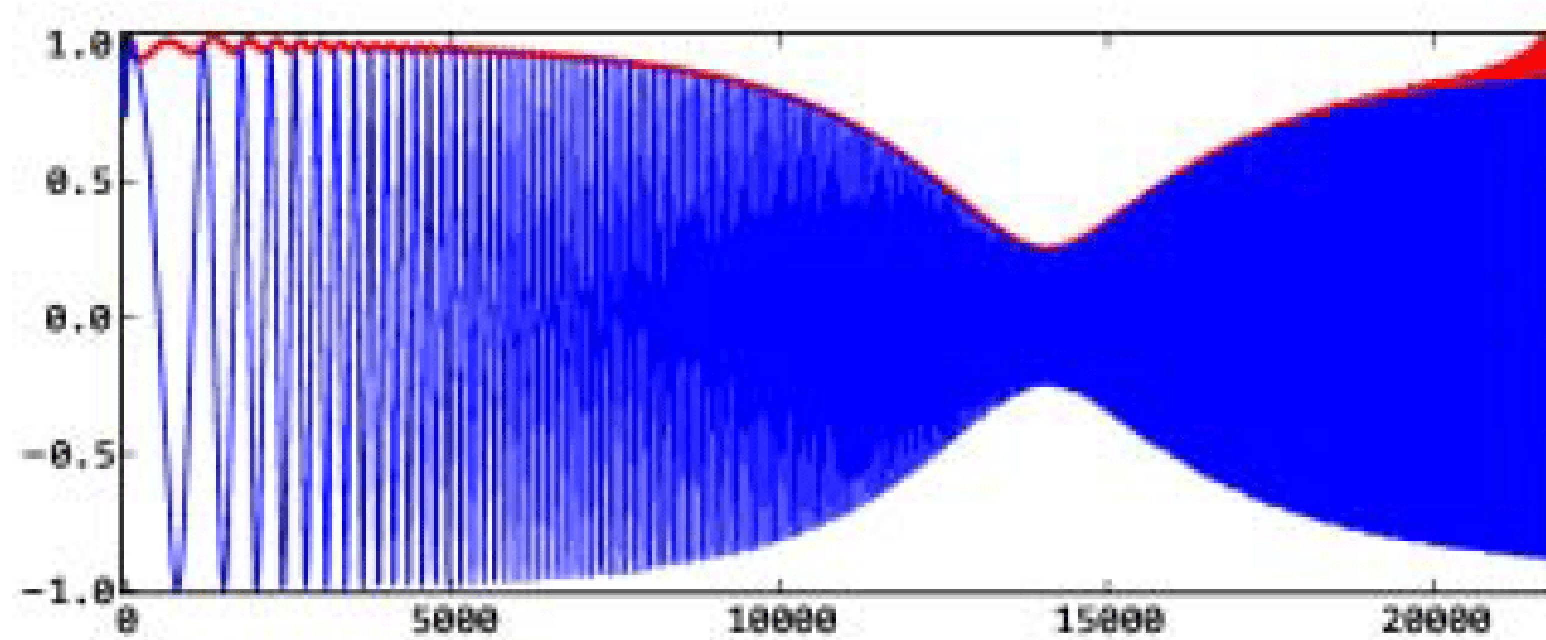


图 15-21　使用 Hilbert 变换对频率扫描波进行包络检波

可以看出，在高频和低频处包络计算出现较大的误差，而在中频部分则能很好地计算出包络的形状。

第 16 章

用C语言提高计算效率

虽然 Python 的开发效率很高，但是它的执行速度却很难满足大量计算的应用需求。因此我们使用 NumPy、SciPy 等库，调用它们已经封装好的高速运算程序，尽量避免直接在 Python 级别进行大量的循环和数值计算。如果这些现成的库无法完成计算要求，就需要用更高效的语言编写核心的计算部分，并为之提供 Python 的调用接口，从而同时实现高效开发和高效运算。

本章介绍几种在 Python 和 C/C++之间相互调用的方法。为了阅读本章的内容，读者需要掌握 C/C++语言的一些相关知识。



如果在编译本章的实例程序时出现错误信息，请在命令行中输入"where gcc"查看 MinGW 编译器的安装路径。如果路径中存在空格或中文，请重新安装 MinGW 编译器到全英文的无空格路径之下。路径中的空格或中文可能会使 Python 无法正确调用它进行编译。

16.1 用 ctypes 调用 DLL 库

ctypes 是 Python 处理动态链接库的标准扩展模块，在 Windows 下使用它可以直接调用 C 语言编写的 DLL 动态链接库。由于它对传递的参数不进行类型和越界检查，因此如果编写的代码有问题，就很可能会造成程序崩溃。将数组数据使用指针传递时，出错的风险将更大。

为了让程序更加安全，通常会用 Python 代码对 ctypes 调用的 DLL 函数进行封装，在调用 DLL 函数之前，在 Python 级别对数据类型和越界进行检查。这样做会使调用接口部分比调用其他的一些手工编写的扩展模块速度慢，但是如果 C 语言函数能一次处理相当多的数据，那么接口调用部分的时耗是可以忽略不计的。

关于 ctypes 的用法，Python 的说明文档里已经有很详细的说明。下面我们着重介绍如何通过 ctypes 让 C 语言函数能处理 NumPy 的数组。

为了方便动态链接库的载入，NumPy 提供了一个便捷函数 ctypeslib.load_library()。它有两个参数，第一个参数是库的文件名，第二个参数是库所在的路径。函数返回的是一个 ctypes 的对象。通过此对象的属性可以访问动态链接库中提供的函数。



ctypes\sum_test.c, sum_test.py

用 SWIG 制作扩展模块，执行“build_dll.bat”就可以用 MinGW 编译 sum_test.c

例如，如果有一个名为“sum_test.dll”的动态链接库，其中提供一个函数 mysum()，其 C 语言程序如下：

```
double mysum(double a[], long n)
{
    double sum = 0;
    int i;
    for(i=0;i<n;i++) sum += a[i];
    return sum;
}
```

可以使用如下语句载入此动态链接库：

```
>>> from ctypes import *
>>> sum_test = np.ctypeslib.load_library("sum_test", ".")
>>> print sum_test.mysum
<_FuncPtr object at 0x037D7210>
```

为了正确调用 mysum()，还需要对其参数类型进行说明，下面的语句描述了 mysum()的两个参数的类型和返回值的类型：

```
>>> sum_test.mysum.argtypes = [POINTER(c_double), c_long]
>>> sum_test.mysum.restype = c_double
```

在上面的代码中，用 POINTER(c_double)声明 mysum()的第一个参数是一个指向双精度浮点数的指针，用 c_long 表示第二个参数的类型是长整型。返回值的类型由 restype 属性指定，这里用 c_double 表示返回值是双精度浮点数类型。

然后就可以调用 mysum()对数组进行求和运算了：

```
>>> x = np.arange(1, 101, 1.0)
>>> sum_test.mysum(x.ctypes.data_as(POINTER(c_double)), len(x))
5050.0
```

这里通过 x.ctypes.data_as(POINTER(c_double))，将数组 x 的地址作为(double *)类型的指针传递给 mysum()。

每次调用 mysum()都进行类型转换是比较麻烦的，因此可以编写一个 Python 函数，将 mysum()封装起来：

```
def mysum(x):
    return sum_test.mysum(x.ctypes.data_as(POINTER(c_double)), len(x))
```

由于数组的元素在内存中存储时可以是不连续的，而且可以是多维数组，因此我们不能指望前面的 mysum()能够处理所有的情况，例如：

```
>>> x = np.arange(1,11,1.0)
>>>  mysum(x[::2]) #对于不连续的数组，结果是错误的
15.0
>>> sum(x[::2])
25.0
```

由于 x[::2]和 x 共用同一块内存空间，而 x[::2]中的元素是不连续的，每两个元素之间的间隔为 16 个字节(两个双精度浮点数大小)。因此若将它传递给 mysum()，实际上计算的是数组 x 中前 5 项的和：1+2+3+4+5=15。而实际上我们希望的结果是：1+3+5+7+9=25。

为了对数组参数进行更详细的描述，NumPy 提供了 ndpointer()。它有 4 个参数——dtype、ndim、shape 和 flags，分别与数组的 4 个同名属性对应。用 ndpointer()对函数的参数进行描述之后，此参数只能接受指定类型的数组，并且能自动将数组转换为对应的指针参数，例如：

```
sum_test.mysum.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float64, ndim=1, flags="C_CONTIGUOUS"),
    c_long
]
```

上面的代码描述了 mysum()的第一个参数是一个类型为双精度浮点数、一维、C 语言连续的数组。在调用 mysum()时，可以直接传递数组，无需再编写一个 Python 函数对其进行封装：

```
>>> sum_test.mysum(x,len(x))
55.0
```

当传递的数组不满足 ndpointer()的指定条件时，就会报错：

```
>>> sum_test.mysum(x[::2],len(x)/2)
ArgumentError: argument 1: <type 'exceptions.TypeError'>:
array must have flags ['C_CONTIGUOUS']
```

由于 x[::2]不是连续的，因此不符合 flags="C_CONTIGUOUS"的设置。

如果我们希望函数能够处理多维、不连续的数组，就需要把数组的 shape 和 strides 属性都传递给它。下面的 mysum2()可以对二维数组的所有元素进行求和：

```
double mysum2(double a[], int strides[], int shapes[])
{
    double sum = 0;
    int i, j, M, N, S0, S1;
    M = shapes[0]; N=shapes[1];
    S0 = strides[0] / sizeof(double); ❶
    S1 = strides[1] / sizeof(double);

    for(i=0;i<M;i++){
```

```
        for(j=0;j<N;j++){
            sum += a[i*S0 + j*S1]; ❷
        }
    }
    return sum;
}
```

mysum2()有三个参数，第一个参数 a 是数组的起始地址，strides 参数是保存数组各个轴元素之间间隔的数组的地址，shapes 参数是保存数组各个轴长度的数组的地址。我们只对二维数组进行处理，因此 strides 和 shapes 数组的长度均为 2。

❶由于 strides 中的数值以字节为单位，因此需要除以 sizeof(double)计算出以双精度浮点数为单位的间隔长度 S0 和 S1。❷这样一来，二维数组 a 中的第 i 行、第 j 列的元素便可以通过 a[i*S0 + j*S1]进行存取。下面用 ctypes 对 mysum2()进行封装：

```
sum_test.mysum2.restype = c_double
sum_test.mysum2.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float64, ndim=2),
    POINTER(c_int),
    POINTER(c_int)
]

def mysum2(x):
    return sum_test.mysum2(x, x.ctypes.strides, x.ctypes.shape)
```

在 Python 的 mysum2()中，为了将数组 x 的 strides 和 shape 属性传递给 C 语言的函数，可以使用 x.ctypes 中提供的 strides 和 shape 属性。注意不能直接传递 x.strides 和 x.shape，因为这些是 Python 的元组对象，而 x.ctypes.shape 得到的是用 ctypes 封装的整数数组：

```
>>> x = np.arange(1, 101, 1.0).reshape((20, 5))
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x020B4DF0>
>>> x.ctypes.shape[0], x.ctypes.shape[1]
(20, 5)
```

可以看出，x.ctypes.shape 是一个 C 语言的包含有两个元素的长整型数组。下面用 mysum2()计算二维数组 x 的所有元素的和，此外还验证了它可以对不连续的数组进行运算：

```
>>> mysum2(x)
5050.0
>>> mysum2(x[::2,::2]) == np.sum(x[::2,::2]) #可以对不连续的数组进行求和
True
```

16.2　用 Weave 嵌入 C++程序

在 3.8 节曾经介绍过，使用 SciPy 库的 Weave 模块可以很方便地将 C++程序以字符串的形式嵌入到 Python 程序中。本节详细介绍如何使用 Weave 库简化 Python 中的 C++程序调用，提高程序的开发和运算速度。

16.2.1　Weave 的工作原理

Weave 能自动对 C++程序进行封装，添加许多编译所需的语句，并将生成的 C++程序保存到某个位置，调用系统中的 C++编译器将其编译为 pyd 文件[①]。使用下面的语句可以获得封装之后的源程序和 pyd 文件的路径：

```
>>> from scipy import weave
>>> weave.catalog.default_dir()
'c:\\docume~1\\...\\locals~1\\temp\\...\\python26_compiled'
```

下面的语句可以获得默认使用的 C++编译器名称：

```
>>> weave.platform_info.choose_compiler()
'msvc'
```

下面的语句调用 C++的 printf()输出变量 a 的内容：

```
>>> a = 100
>>> weave.inline(r'printf("%d\n", a);', ["a"], compiler="gcc")
100
```

如果读者的 Windows 系统中安装了 Visual C++编译器，那么 Weave 默认会选择它作为编译器。但使用'msvc'作为编译器会遇到许多麻烦，因此建议读者采用和 Python(x,y)一起安装的 gcc 编译器(MinGW)，只需要设置 compiler 参数为"gcc"即可。

由于第一次运行时需要调用 C++编译器进行编译，因此要等几秒钟才能看到运行结果。下面我们看看 Weave 是如何对 printf()进行封装的。打开 weave.catalog.default_dir()目录中最新创建的 cpp 文件。此文件有 700 余行代码，在其中搜索“printf”就可以找到调用 printf()的代码，如下所示：

```
static PyObject* compiled_func(PyObject*self, PyObject* args)
{
    ...
    PyObject *py_a;
```

① pyd 文件是 Python 的动态链接库模块，在 Windows 下它和 DLL 文件是一样的。

```
    ...
        PyObject* raw_locals __attribute__ ((unused));
        raw_locals = py_to_raw_dict(py__locals,"_locals");  ❶
        PyObject* raw_globals __attribute__ ((unused));
        raw_globals = py_to_raw_dict(py__globals,"_globals"); ❶
        /* argument conversion code */
        py_a = get_variable("a",raw_locals,raw_globals); ❷
        int a = convert_to_int(py_a,"a"); ❸
        /* inline code */
        /* NDARRAY API VERSION 1000009 */
        printf("%d\n", a);  ❹

    ...
}
```

我们只需要大致了解上述程序的结构，没有必要深究每条语句是如何运行的。

❶首先通过一些语句得到两个 PyObject 对象——raw_locals 和 raw_globals，它们相当于 Python 内置函数 locals()和 globals()的返回值，是两个 Python 的字典对象，Python 变量 a 的值就在 raw_globals 中。

❷通过调用 get_variable()从 raw_locals 或 raw_globals 中获得变量 a 的值，获得的是一个 Python 的整数对象，因此用 py_a 指针指向它。

❸通过调用 convert_to_int()获取 py_a 对象中的整数值，并赋值给一个 C 语言的整型变量 a，这样就完成了从 Python 变量到 C 变量的转换。

❹传递给 inline()的 C 语言代码，将调用 printf()打印出 C 语言变量 a 的值，也就是 Python 变量 a 的值。

从上面的程序可以看出，在 C 语言程序中所有以“py_”开头的变量都是 Python 对象，而和 Python 变量同名的变量则是调用类型转换函数(convert_to_int 等)得到的结果。在调用 inline()时，Weave 会根据 Python 变量的值的类型，输出不同的代码。由于前面已经编译过 a 为整数时的程序了，因此再次执行时不需要重复进行编译：

```
>>> a = 200
>>> weave.inline(r'printf("%d\n", a);', ["a"], compiler="gcc")
```

如果将 a 改为浮点数，那么 Weave 就需要重新编译程序了：

```
>>> a = 200.0
>>> weave.inline(r'printf("%d\n", a);', ["a"], compiler="gcc")
0
```

由于 printf()中使用“%d”格式输出数值，因此无法正确输出浮点数变量 a 的值。编译输出目录下会增加一个 cpp 文件，其中的变量类型转换语句变成了：

```
double a = convert_to_float(py_a,"a");
```

inline()有许多关键字参数可以改变它所输出的 C 语言程序。例如，通过 headers 参数可以指定程序所需的头文件列表，通过 support_code 参数可以设置主程序之外的代码，另外还可以定义主程序中使用的函数、全局变量、结构体以及类等代码。具体的用法请读者参考 inline()的使用手册。

Weave 安装路径下的 examples 文件夹中有许多嵌入 C++程序的实例，读者可以通过运行这些例子，并查看扩展之后的 C++源程序，以进一步深入了解 Weave 的工作原理。

16.2.2 处理 NumPy 数组

Weave 能够自动在 Python 类型和 C++类型之间进行转换。要编写处理 NumPy 数组的 C++程序，首先要弄清楚 Weave 是如何将数组传递给 C++语言的。下面的语句创建一个元素类型为长整型的二维数组 a，并执行 weave.inline()得到扩展之后的 C++源代码：

```
>>> import numpy as np
>>> a = np.arange(12).reshape(-1,4)
>>> weave.inline("//dummy",["a"], compiler="gcc")
```

在自动扩展之后的 C++程序中，进行类型转换的代码如下：

```
/* argument conversion code */
py_a = get_variable("a",raw_locals,raw_globals);
PyArrayObject* a_array = convert_to_numpy(py_a,"a");
conversion_numpy_check_type(a_array,PyArray_LONG,"a");
#define A1(i) (*((long*)(a_array->data + (i)*Sa[0])))
#define A2(i,j) (*((long*)(a_array->data + (i)*Sa[0] + (j)*Sa[1])))
#define A3(i,j,k) (*((long*)(a_array->data + (i)*Sa[0] + (j)*Sa[1] + (k)*Sa[2])))
#define A4(i,j,k,l) //... 太长，省略
npy_intp* Na = a_array->dimensions;
npy_intp* Sa = a_array->strides;
int Da = a_array->nd;
long* a = (long*) a_array->data;
```

对于 NumPy 数组 a，通过类型转换程序之后得到如下可用的 C++变量：

- py_a：PyObject 类型的指针。
- a_array：PyArrayObject 类型的指针。
- Na：指向 a.shape 数据的指针，npy_intp 实际上是某种整数类型，它决定数组每个轴的长度。
- Sa：指向 a.strides 数据的指针，它决定数组某个轴上元素之间的间隔(以字节为单位)。
- Da：其内容为 a.ndim，即数组的维数(轴数)。
- a：数组的元素类型的指针，指向数组 a 的数据区。

其中的 py_a 和 a_array 是 Python 对象，直接使用它们得不到特别的速度提升，因此在 C++ 程序中需要通过 Na、Sa、Da 和 a 等几个变量读写数组。为了通过指针 a 访问数组中特定下标的元素，需要将元素的下标转换为相对于指针 a 的偏移量。为了方便程序的编写，Weave 定义了 4 维以下数组的元素访问宏：A1、A2、A3 和 A4。宏中使用下标和 Sa 的值计算出元素的偏移量，注意这里的偏移量是以字节为单位的，因此在宏中不直接使用变量 a 进行地址计算，而是使用 a_array->data，因为它是类型为(char *)的指针。

下面的程序输出这些信息：



weave\weave_numpy_info.py
在 C++程序中输出 NumPy 数组的信息

```
from scipy import weave
import numpy as np

def print_array_info(arr):
    weave.inline(r"""
    int i;
    printf("arr.ndim=%d\n", Darr);
    printf("arr.shape=");
    for(i=0;i<Darr;i++)
    {
        printf("%d ", Narr[i]);
    }
    printf("\n");
    printf("arr.strides=");
    for(i=0;i<Darr;i++)
        printf("%d ", Sarr[i]);
    printf("\n");
    """, ["arr"], compiler="gcc")

a = np.arange(12).reshape(-1, 4)
print_array_info(a)
print_array_info(a[:,::2])
```

此程序的输出如下：

```
arr.ndim=2
arr.shape=3 4
arr.strides=16 4

arr.ndim=2
arr.shape=3 2
arr.strides=16 8
```

Weave 还可以将 NumPy 数组转换为 Blitz++数组，从而简化 C++程序的编写工作。只需要给 inline()传递一个 type_converters 关键字参数即可。

Blitz++

Blitz++是一个 C++的科学计算类库，它大量使用模板技术来实现高效率运算，效率可以和 Fortran 语言相媲美。



http://www.oonumerics.org/blitz
Blitz++的官方网址

下面的语句使用 Blitz++数组对 NumPy 数组 a 进行封装，为了和前面的程序相区别，这里我们先将数组 a 转换为一个三维数组：

```
>>> a.shape = 2,2,3
>>> weave.inline(r'//',["a"],compiler="gcc",type_converters=weave.converters.blitz)
```

让我们看看扩展之后的 C++源程序：

```
py_a = get_variable("a",raw_locals,raw_globals);
PyArrayObject* a_array = convert_to_numpy(py_a,"a");
conversion_numpy_check_type(a_array,PyArray_LONG,"a");
conversion_numpy_check_size(a_array,3,"a");
blitz::Array<long,3> a = convert_to_blitz<long,3>(a_array,"a");
blitz::TinyVector<int,3> Na = a.shape();
```

我们得到了下面两个 C++变量，它们的类型在 Biltz++库中定义：

- a：类型为 blitz::Array<long,3>，它是一个元素类型为 long 的三维数组，通过它可以存取数组中的元素。
- Na：类型为 blitz::TinyVector<int,3>，它是一个元素类型为 int 的长度为 3 的向量，通过它可以获得数组每个轴的长度。

介绍 blitz::Array 的用法已经超出了本书的范围，读者可以自行阅读其文档，下面只简单介绍如何存取数组中的元素。

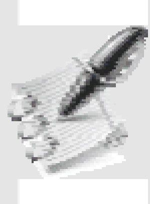
blitz 数组类覆盖了“()”操作符作为元素存取用，因此只需要将一系列整数传递给()操作符即可。例如，A 是一个 blitz 数组对象，下面的语句可以对其中下标为(7,1,0)的元素赋值：

```
A(7,1,0) = 5;
```

也可以使用 TinyVector 类型的变量存取数组中的元素，由于数组 A 是三维的，因此 index 向量的长度也必须是 3：

```
TinyVector<int, 3> index;
index = 7, 1, 0;
A(index) = 5;
```

下面的程序给数组顺序填写上递增序列：



weave\weave_numpy_blitz.py
将 NumPy 数组转换为 Blitz++数组

```
from scipy import weave
import numpy as np

def set_array(arr):
    weave.inline(r"""
    int i, j, k;
    double v = 0.0;
    for(i=0;i<Narr(0);i++)
    for(j=0;j<Narr(1);j++)
    for(k=0;k<Narr(2);k++)
    {
        arr(i,j,k) = v;
        v += 1.0;
    }
    """, ["arr"], compiler="gcc", type_converters=weave.converters.blitz)

a = np.zeros(12).reshape(2,2,3)

set_array(a)
print a
```

程序的输出为：

```
[[[  0.   1.   2.]
  [  3.   4.   5.]]

 [[  6.   7.   8.]
  [  9.  10.  11.]]]
```

16.2.3　使用 blitz()提速

NumPy 的数组运算虽然是在 C 语言级别进行循环的，但是它的每个表达式在运算时都会产生一个新的数组。因此在进行大数组运算时，如果表达式很复杂，就会产生许多巨大的临时数组，降低运算速度。

weave.blitz()可以将 NumPy 的数组运算表达式编译为 Blitz++的数组运算程序，从而提高数

组的运算速度。下面的例子计算图像的浮雕效果，每次运算都会产生 3 个临时数组：

```
>>> import pylab as pl
>>> import scipy as sp
>>> p = sp.lena()
>>> p.shape
(512, 512)
>>> e1 = p[:-2,1:-1] - p[2:,1:-1] + p[1:-1,:-2] - p[1:-1,2:]
```

用 weave.blitz()可以提高运算速度并减少内存的使用。由于 blitz()不能产生新的数组，因此首先要初始化一个数组用来保存结果：

```
>>> e2 = np.zeros((p.shape[0]-2, p.shape[1]-2), np.int32)
```

然后将 NumPy 的数组运算表达式当做字符串传递给 blitz()。第一次运算时由于需要进行编译，会等上几秒钟：

```
>>> weave.blitz("e2 = p[:-2,1:-1]-p[2:,1:-1]+p[1:-1,:-2]-p[1:-1,2:]")
>>> np.all(e1==e2)
True
```

读者可能会怀疑 blitz()的运算速度是否真的会比 NumPy 快，我们用下面的程序比较几种计算方法的速度，其中包括使用 np.add()等函数防止临时数组变量的产生。程序中为了使比较结果更加明显一些，调用 ndimage.zoom()将图像放大一倍，并且将其元素转换为浮点数，在调用 timeit()测试时间之前，先调用一次 blitz()，使其预先将 C++程序编译好。



weave\weave_blitz_speed.py
比较 blitz()和 NumPy 的数组运算速度

```
import numpy as np
import scipy as sp
from timeit import timeit
from scipy import weave, ndimage

def numpy01(p):
    return p[:-2,1:-1] - p[2:,1:-1] + p[1:-1,:-2] - p[1:-1,2:]

def numpy02(p):
    e1 = np.zeros((p.shape[0]-2, p.shape[1]-2), np.float)
    np.subtract(p[:-2,1:-1], p[2:,1:-1] , e1)
    np.add(e1, p[1:-1,:-2], e1)
    np.subtract(e1, p[1:-1,2:])
    return e1
```

```
def blitz(p):
    e2 = np.zeros((p.shape[0]-2, p.shape[1]-2), np.float)
    weave.blitz("e2 = p[:-2,1:-1]-p[2:,1:-1]+p[1:-1,:-2]-p[1:-1,2:]")
    return e2

p = sp.lena()
p = ndimage.interpolation.zoom(p, 2).astype(np.float)
blitz(p)

if __name__ == "__main__":
    import_str = "from __main__ import numpy01, numpy02, blitz, p"
    for s in ["numpy01", "numpy02", "blitz"]:
        print s, timeit(s+"(p)", import_str, number = 100)
```

程序的输出为：

```
numpy01 2.35462162538
numpy02 2.21472903504
blitz 1.04954585395
```

可以看出，调用 np.add()等函数只有微小的速度提升，而使用 blitz()则能够提速一倍。

16.2.4 扩展模块

weave.inline()和 weave.blitz()都自动创建扩展模块，在内部，它们使用 weave.ext_tools 模块中提供的类创建扩展模块。我们也可以直接使用 ext_tools 提供的工具，这样创建的扩展模块将保存到程序运行的当前文件夹中，而不是临时文件夹。

下面的程序使用 ext_tools 创建一个名为 demo_ext 的扩展模块，其中有一个 square()，它将二维数组中的每个元素设置为其平方值：



weave\weave_ext.py
使用 weave 手工创建扩展模块

```
from scipy import weave
import numpy as np

def build_ext():
    mod = weave.ext_tools.ext_module('demo_ext', compiler="gcc") ❶

    ext_code = r"""
    int i, j;
    for(i=0;i<Narr(0);i++)
    for(j=0;j<Narr(1);j++)
    {
```

```
        arr(i,j) *= arr(i,j);
    }
    """
    arr = np.zeros((2,2))
    func = weave.ext_tools.ext_function('square',ext_code,['arr'], ❷
         type_converters=weave.converters.blitz)
    mod.add_function(func) ❸

    mod.compile()

if __name__ == "__main__":
    try:
        import demo_ext
    except ImportError:
        build_ext()
        import demo_ext

    a = np.arange(0, 100, 1.0).reshape((10, 10))
    demo_ext.square(a)
    print a
```

❶首先使用 ext_module()创建一个扩展模块 mod。❷然后使用 ext_function()将 C++程序封装成扩展函数 func。在调用 ext_function()之前，创建了一个 2*2 的临时数组 arr，这样才能根据 arr 的类型生成正确的 C++程序。❸最后调用 mod 的 add_function()将函数添加到扩展模块中。可以用这种方法为扩展模块添加多个函数。

最后在主程序中，先尝试载入已经生成的扩展模块 demo_ext。如果载入失败，就调用 build_ext()创建扩展模块，然后再载入它。

16.3 用 Cython 将 Python 编译成 C

Cython 是为了减轻使用 C 语言开发 Python 扩展模块的负担而开发出来的一种编程语言。它的语法基本上和 Python 相同，但增加了直接定义和调用 C 语言函数、定义变量类型等功能。通过 Cython 的编译器可以将 Cython 的源程序编译成 C 语言的源程序，再通过 C 语言编译器将源程序编译成扩展模块。



http://cython.org
Cython 的官方网址

16.3.1 编译 Cython 程序

为了使用 Cython，首先要选择好 C 语言编译器。在 Windows 下可以选择 VC++或 MinGW，

为了让 Python 使用 MinGW 作为默认的编译器，请先编辑[②]文件：

```
c:\Python26\lib\distutils\distutils.cfg
```

在其中添加如下内容：

```
[build]
compiler = mingw32
```

Cython 的程序文件以.pyx 为扩展名，为了编译它，需要编写一个“setup.py”文件：



cython\setup.py
编译 Cython 程序用的配置文件

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

ext_modules = [
    Extension("mylib", ["mylib.pyx"],
        include_dirs = [numpy.get_include(),'.']),
    Extension("cython_matmul", ["cython_matmul.pyx"],
        include_dirs = [numpy.get_include(),'.'])
]

setup(
  name = 'my cython library test',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

在此程序中，指定编译的文件为“mylib.pyx”和“cython_matmul.pyx”，并且包括 NumPy 的 C 语言头文件[③]。编译时只需要运行：

```
setup.py build_ext --inplace
```

就可得到编译后的扩展模块文件“mylib.pyd”和“cython_matmul.pyd”。

16.3.2 提高计算效率

为了评价 Cython 的计算效率，下面编写一组函数，计算数组的正弦平方和，并通过 timeit

② 如果“distutils.cfg”文件不存在，新建一个。
③ 后面我们会看到如何使用 Cython 处理 NumPy 的数组对象。

模块测量它们的计算时间。下面是这个测试程序：



cython\cython_test.py

评价各种计算“正弦平方和”的函数的运算速度

```
from timeit import timeit
from math import sin
import mylib as cy

def py_test1(xs):
    s = 0.0
    for x in xs:
        s += sin(x)**2
    return s

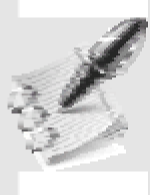
def py_test2(xs):
    return sum(sin(x)**2 for x in xs)

x = [t*0.01 for t in range(10000)]

if __name__ == "__main__":
    s = "from __main__ import x, py_test1, py_test2, cy"

    for f in ["py_test1", "py_test2", "cy.test1", "cy.test2", "cy.test3"]:
        print f, timeit("%s(x)" % f, s, number=100)
```

程序中定义了两个 Python 函数——py_test1 和 py_test2，然后载入用 Cython 编写的 mylib 库，测试其中的 test1、test2 和 test3 三个函数的计算时间。



cython\mylib.pyx

用 Cython 计算列表中各个元素的正弦值的平方和

```
from math import sin as pysin
cdef extern from "math.h":
    double sin(double x)

def test1(xs):
    s = 0.0
    for x in xs:
        s += pysin(x)**2
    return s

def test2(xs):
    s = 0.0
```

```
        for x in xs:
            s += sin(x)**2
        return s

    def test3(xs):
        cdef double s = 0.0
        for x in xs:
            s += sin(x)**2
        return s
```

程序的运行结果如下：

```
py_test1 0.378925665182
py_test2 0.393813566603
cy.test1 0.379718392833
cy.test2 0.0735996653795
cy.test3 0.0428850451455
```

用 Cython 优化后的计算速度得到将近 9 倍的提升。下面逐个分析“mylib.pyx”中的三个函数。

首先，test1()和 Python 版本的 py_test1 完全一样，使用 pysin 作为函数名是为了和后面的 C 语言的 sin 函数相区别。而进行编译之后没有得到任何的效率提升，这是因为它完全使用的是 Python 的东西：变量 s 是一个 Python 对象，pysin 是一个 Python 函数。如果读者有耐心查看“mylib.pyx”编译后的“mylib.c”程序，就可以找到“s += pysin(x)**2”编译之后的一大堆 C 程序。这里从中摘出和 pysin 函数调用相关的两条语句，可以看出由于 pysin 是一个 Python 函数，因此，首先需要通过__Pyx_GetName 获得它，然后再通过 PyObject_Call 调用它：

```
__pyx_2 = __Pyx_GetName(__pyx_m, __pyx_kp_pysin);
...
__pyx_t_4 = PyObject_Call(__pyx_2, ((PyObject *)__pyx_t_3), NULL);
```

在 test2()中，我们调用 C 语言标准库中的 sin 函数，C 语言标准库中的函数使用如下的语法进行声明：

```
cdef extern from "math.h":
    double sin(double x)
```

在“mylib.c”中可以找到它编译之后的语句，注意这里直接调用 C 语言版本的 sin()：

```
__pyx_t_3 = PyFloat_FromDouble(pow(sin(__pyx_t_4), 2));
```

由于在 test1()和 test2()中没有说明变量 s 的类型，因此在输出的 C 语言程序中，它是一个 Python 对象，它的声明及求和计算被编译为如下语句，通过调用 PyNumber_InPlaceAdd 实现加

法运算：

```
PyObject *__pyx_v_s;
...
PyNumber_InPlaceAdd(__pyx_v_s, __pyx_t_3);
```

在 test3()中，使用下面的语句直接定义了一个 C 语言的双精度类型的变量来保存求和结果：

```
cdef double s = 0.0
```

于是它被编译成：

```
double __pyx_v_s;
...
  __pyx_v_s += pow(sin(__pyx_t_4), 2);
```

到这里，循环体内部的运算已经足够快了，但是由于传入的参数是一个 Python 的列表对象，即使是 C 语言级别，对列表对象进行循环计算的效率也仍然只能是 Python 级别的。为了进一步进行优化，我们希望在 C 语言级别对数组进行循环，显然直接处理 NumPy 数组是提高速度的关键所在。

16.3.3 快速访问 NumPy 数组

Cython 支持快速访问 NumPy 的数组对象。为了优化使用 NumPy 数组的程序，必须在程序最前面使用 cimport 载入 NumPy 的 pxd 文件：

```
cimport numpy as np
```

Cython 中的 pxd 文件相当于 C 语言的头文件，它包含 Cython 的定义，使用 cimport 关键字载入。读者可以在下面的文件夹下找到 Cython 自带的 pxd 头文件：

```
c:\Python26\Lib\site-packages\Cython\Includes
```

下面通过逐步优化矩阵乘积[④]的程序，介绍在 Cython 中如何快速存取 NumPy 数组中的元素。



cython\cython_matmul.pyx，cython_test_matmul.py
用 Cython 计算矩阵乘积及其测试程序

首先，下面的 matmul1()是在 Python 级别进行循环的矩阵乘积函数，将它原封不动地使用 Cython 编译成扩展模块之后，对 100*100 的两个矩阵进行乘积运算，其运算时间为 1.3766 秒。

④ 这里只是作为例子测试 Cython 的速度，真正的矩阵乘积运算请使用 np.dot()计算。例如，本例中 dot()的计算时间为 0.001 秒，比 Python 版本快 1500 倍。

直接在 Python 下运行的时间为 1.6183 秒，只比 Python 版本快不到 1.2 倍。

```
def matmul1(A, B, out):
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
                out[i,j] = s
```

使用前面介绍的 cdef 关键字定义 C 语言的局域变量能提高运算速度。下面将几个循环变量都定义为 int 类型，把求和用的临时变量 s 定义为 double 类型。因为它们都是 C 语言的变量，因此计算时间缩短为 0.98 秒，速度是纯 Python 版本的 1.6 倍。

```
def matmul2(A, B, out):
    cdef int i, j, k
    cdef double s
    ...
```

前面两个函数没有告诉编译器如何快速访问数组，因此只能使用 Python 提供的方法来访问。最耗时的数组存取部分没能得到优化，所以计算速度得不到数量级的改进。下面通过声明三个参数数组的元素类型和维数，使 Cython 知道如何存取数组的元素，从而大幅度提高计算速度。matmul3()的计算时间为 0.0127 秒，速度是纯 Python 版本的 130 倍。

```
def matmul3(np.ndarray[np.float64_t, ndim=2] A,
            np.ndarray[np.float64_t, ndim=2] B,
            np.ndarray[np.float64_t, ndim=2] out):
    ...
```

这里通过一种特殊的 Cython 语法声明高效的数组存取变量。它告诉 Cython 编译器：A、B 和 out 都是元素类型为 np.float64_t 的二维数组，其中的 float64_t 在“numpy.pxd”中定义。关于此语法的详细解释请参考下面的链接。



http://wiki.cython.org/enhancements/buffer
介绍如何使用 Cython 高效存取数组

如果读者仔细查看编译后的“cython_matmul.c”，就会发现简单的一行“out[i,j]=s”被编译为 16 行 C 语言程序，其中绝大部分都是在检测下标变量 i 和 j 是否超出了 out 数组的范围以及处理下标为负数的情况。显然这种处理对于矩阵乘积的程序毫无意义，因此可以使用 boundscheck 和 wraparound 两个修饰，告诉编译器关闭负数下标处理和越界检查。matmul4()的运算时间为 0.0044 秒，速度是纯 Python 版本的 360 倍。

```
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def matmul4(np.ndarray[np.float64_t, ndim=2] A,
           np.ndarray[np.float64_t, ndim=2] B,
           np.ndarray[np.float64_t, ndim=2] out):
           ...
```

虽然使用 Cython 对数组存取进行优化，可以提高几百倍的计算速度，但是要注意只有以下两个条件满足时，通过 Cython 才能对数组存取进行优化：

- 下标数必须和数组的维数一样多。
- 所有下标必须是 C 语言的整型变量或整数。

因此，A[i][j]、A[i,:]等都不能被优化。并且如果变量 i 不使用 cdef 声明，A[i,0]将不能被优化。

16.4 用 SWIG 创建扩展模块

除了使用 Weave 和 Cython 自动创建扩展模块之外，我们还可以使用 SWIG 为现有的 C/C++源程序创建 Python 的调用接口，从而将其封装成能够被 Python 使用的扩展模块。SWIG 的英文全称为 Simplified Wrapper and Interface Generator，直接翻译成中文就是“简单接口生成器”。它是一个为 C/C++程序创建各种动态语言的接口的开发工具，本节介绍如何使用 SWIG 为 Python 制作 C/C++编写的扩展模块。

16.4.1 SWIG 的调用方法和实例

假设有现成的 C 语言程序——“demo.h”和“demo.c”，我们想在 Python 中使用上述程序中定义的变量或函数，可以按照下面的步骤进行，整个过程如图 16-1 所示。

- 为了告诉 SWIG 需要将 C 语言程序中的哪些函数和变量输出到 Python 中，需要编写一个扩展名为“.i”的接口文件，图 16-1 中的文件名为“demo.i”。
- 调用 SWIG 将接口文件转换为 C 和 Python 的两个源程序文件，图 16-1 中 SWIG 通过“demo.i”生成“demo_wrap.c”和“demo.py”。
- 调用 C 语言的编译器将“demo.h”、“demo.c”和“demo_wrap.c”三个文件编译成 Python 的扩展模块“_demo.pyd”。
- 用户的 Python 程序可以直接调用“_demo.pyd”模块，或者通过“demo.py”间接调用它。

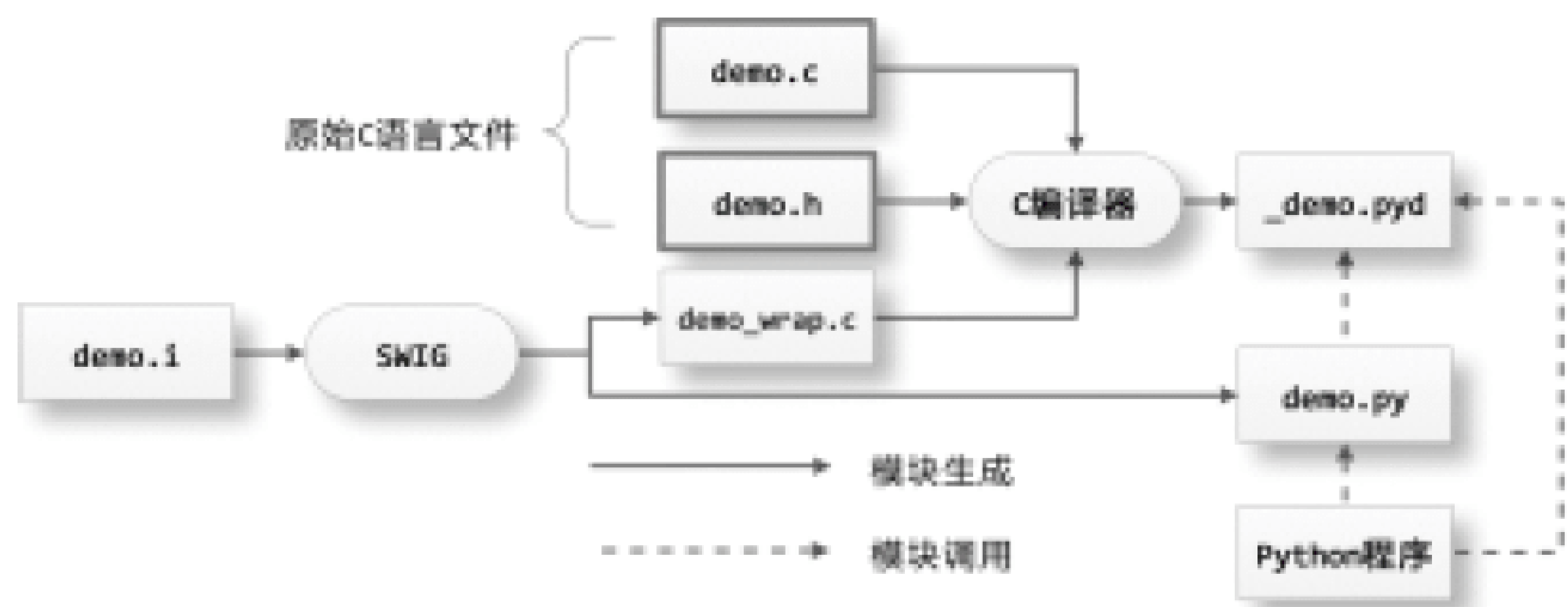


图 16-1　使用 SWIG 创建扩展模块的流程

通过 Python 的标准模块 distutils 可以将上述编译过程自动化，我们只需要编写一个“setup.py”程序，在其中设置好编译扩展模块所需的文件，就可以实现从源文件到扩展模块文件的全自动编译。下面让我们看一个例子。

swig_demo\demo.cpp, demo.h, demo.i, setup.py
用 SWIG 制作扩展模块

如果希望使用 MinGW 作为 C 语言编译器，请按照 16.3.1 节中的介绍进行配置。

在“demo.c”中定义了两个函数，在“demo.h”中有这两个函数的调用声明：

```
double power(double x);
double sum_power(int n);
```

我们希望在 Python 中能调用其中的 sum_power()，于是编写如下的“demo.i”文件：

```
%module demo

%{
#include "demo.h"
%}

double sum_power(int n);
```

在“demo.i”中，以“%”开头的行对 SWIG 有特殊的意义。第一行的%module 命令告诉 SWIG 生成一个名为 demo 的扩展模块。以“%{”和“%}”括起来的部分将原封不动地输出到“demo_wrap.cpp”中，请读者在生成的“demo_wrap.cpp”中搜索“demo.h”试试看。最后一行是和头文件中一样的函数声明，SWIG 将对接口文件中声明的所有函数进行封装，使它们可以在 Python 中使用。因此对于：

```
double sum_power(int n);
```

将在“demo_wrap.cpp”中出现一个如下所示的封装函数，也请读者自行搜索此函数，仔细分析封装函数的源代码，以了解 SWIG 的工作原理：

```
SWIGINTERN PyObject *_wrap_sum_power(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
    // ... 省略
    result = (double)sum_power(arg1);
    // ... 省略
}
```

下面是自动编译用的“setup.py”文件：

```
from distutils.core import *

demo_module = Extension(
    '_demo',
    ['demo.i', 'demo.cpp'],
    swig_opts=["-c++"]
)

setup(
    name = 'demo',
    version = '0.1',
    author      = "HYRY Studio",
    description = """Simple swig demo""",
    ext_modules = [demo_module],
    py_modules = ["demo"]
)
```

这段程序通过 Extension()创建了一个扩展模块_demo，它由“demo.i”和“demo.cpp”两个文件构成。虽然“demo.cpp”完全是 C 语言程序，但是为了演示如何编译 C++程序，这里通过 swig_opts 关键字将参数“-c++”传递给 SWIG。

在命令行中运行下面的命令，就会产生 SWIG 的封装文件“demo_wrap.cpp”，并且调用编译器生成扩展模块“_demo.pyd”：

```
setup.py build_ext --inplace
```

“_demo.pyd”虽然可以直接在 Python 中使用，但是使用 SWIG 提供的“demo.py”更方便一些：

```
>>> import demo
>>> demo.sum_power(10)
285.0
```

demo 模块实际上是由“demo.py”定义的：

```
>>> demo
<module 'demo' from 'demo.py'>
```

而编译所得的扩展模块则可以通过 demo._demo 获得：

```
>>> demo._demo
<module '_demo' from '_demo.pyd'>
```

16.4.2 SWIG 基础

SWIG 对接口文件中声明的函数和变量进行封装，自动生成可在 Python 中调用的函数。在封装函数中主要进行下面 3 项工作：

- 将封装函数接收到的 Python 对象转换为调用 C/C++函数所需的值。
- 调用相应的 C/C++函数。
- 将 C/C++函数的返回值再转换为 Python 对象。

由于 Python 和 C/C++支持的数据类型有很大区别，因此有时这种转换工作是比较麻烦的。为了完全理解 SWIG 的工作原理，请读者认真阅读 SWIG 的文档。本节只对一些比较常见的应用进行简单的说明。



swig_basic\demo.cpp, demo.h, demo.i, setup.py
SWIG 的基础用法演示

1. 指针

SWIG 支持 C/C++的指针类型，C/C++的指针在 Python 中用一种特殊的对象表示。例如在接口文件中，写入如下标准库中的文件操作函数声明：

```
FILE *fopen(const char *filename, const char *mode);
int fputs(const char *, FILE *);
int fclose(FILE *);
```

在 Python 中可以直接调用这些函数：

```
>>> import demo
>>> f = demo.fopen("test.txt", "w")
>>> f
<Swig Object of type 'FILE *' at 0x0149D8D8> # 指针在 Python 中是 SwigPyObject 对象
>>> demo.fputs("test file\n", f)
0
>>> demo.fclose(f)
0
```

在 Python 中，用 SwigPyObject 类型的对象保存 C 语言的指针，该对象记录下了指针的类型和地址，以便将其传回 C 语言函数时使用。这些信息对于 Python 语言来说几乎是没有用的，因此在 Python 中我们无法通过变量 f 获得 FILE 结构体中的内容。

由于 C 语言不区分数组和指针，因此如果只是用 SwigPyObject 类型对 C 语言的数组进行封装，那么在 Python 中将无法通过它存取数组中的元素。为了解决这个问题，可以编写专门存取数组元素的函数。假设用 C 语言编写了下面 4 个函数，并通过 SWIG 将它们都制作成扩展模块：

```
double * make_array(int n); // 分配一个包含 n 个元素的双精度数组
void free_array(double * x); // 释放数组
double get_element(double * x, int n); // 获得数组 x 的第 n 个元素
void set_element(double * x, int n, double v) // 设置数组 x 的第 n 个元素
```

在 Python 中可以使用 set_element()和 get_element 存取数组的元素：

```
>>> import demo
>>> a = demo.make_array(10) # 创建一个包含 10 个元素的数组
>>> demo.set_element(a, 5, 1.2) # 相当于 a[5] = 1.2
>>> demo.get_element(a, 5) # 相当于 a[5]
1.2
>>> demo.free_array(a) # 释放数组
```

如果没有定义 get_element()和 set_element()，也可以使用 ctypes 的 cast()将 SWIG 指针对象转换为可以用下标存取的数组。首先，通过 SwigPyObject 对象的__long__()方法获得其指向的地址值：

```
>>> a = demo.make_array(10)
>>> a.__long__() # 指针对象指向的地址
28807064
```

调用 ctypes 的 cast()可以将地址转换为 ctypes 的指针对象，它的第一个参数为地址值，第二个参数为要转换成的数据类型。下面语句中的数据类型为 ctypes.POINTER(ctypes.c_double)，表示要转换为一个指向 double 类型的指针：

```
>>> import ctypes
>>> ca = ctypes.cast(a.__long__(), ctypes.POINTER(ctypes.c_double))
>>> demo.set_element(a, 3, 2.0) # 通过 SWIG 封装函数设置第三个元素
>>> ca[3] # 通过 ctypes 指针获得第三个元素
2.0
>>> ca[2] = 100 # 通过 ctypes 指针设置第二个元素
>>> demo.get_element(a, 2) # 通过 SWIG 封装函数获得第二个元素
100.0
```



内存的分配和释放必须由用户程序完成，并且没有数组越界检测。在 Python 中，使用封装函数和直接使用 C 语言的函数一样存在越界的危险。

显然这种数组访问方式完全没有 Python 的风格，但是它是最基本的解决方案。后面我们还将详细介绍如何通过 SWIG 实现用 C 语言程序处理 NumPy 数组。

2. 全局变量

SWIG 不但可以对 C 语言的函数进行封装，还可以对全局变量进行封装，使得它们的内容可以在 Python 中进行读写。例如，在“demo.cpp”中定义了一个全局变量 global_test 以及输出其内容的函数 print_global()：

```
double global_test = 100.0;

void print_global()
{
    printf("global_test=%f\n", global_test);
}
```

在“demo.h”中，对变量和函数进行声明：

```
extern double global_test;
void print_global();
```

然后在接口文件“demo.i”中，用下面的语句将 global_test 和 print_global()输出到 Python 的扩展模块中：

```
double global_test;
void print_global();
```

在生成的扩展模块中，所有的全局变量都通过模块的 cvar 属性进行存取：

```
>>> import demo
>>> demo.cvar.global_test # 查看全局变量的内容
100.0
>>> demo.cvar.global_test *= 3 # 修改全局变量的内容
>>> demo.print_global() # 用 C 语言的函数输出全局变量的内容
global_test=300.000000
```

3. 结构和类

SWIG 会自动对结构或类中的每个成员变量添加存取函数，并输出到扩展模块中。在 SWIG 输出的 Python 模块[5]中，再对扩展模块中的函数进行封装，让 C 语言的结构类型能够像 Python

⑤ 在本例中，SWIG 输出的模块的文件名为“demo.py”。

的类一样使用。例如，将下面的 C 语言结构声明添加到接口文件“demo.i”中：

```
struct Point
{
    double x, y;
};
```

在 Python 中，会将 Point 结构封装成一个 Point 类，它的每个属性都与结构中的每个成员变量相对应：

```
>>> import demo
>>> a = demo.Point()
>>> a.x
0.0
>>> a.y
0.0
>>> a.x = 100
>>> a.y = 200
>>> a.x
100.0
>>> a.y
200.0
```

也可以直接调用扩展模块中的函数：

```
>>> demo._demo.Point_x_get(a)
100.0
```

实际上，如果读者阅读“demo.py”中的程序就会发现：Point 内属性 x、y 的存取都会调用 Point_x_get、Point_x_set、Point_y_get、Point_y_set 等函数。而在“demo_wrap.cpp”中，可以找到它们对应的 C 语言函数：

```
SWIGINTERN PyObject *_wrap_Point_x_get(PyObject *SWIGUNUSEDPARM(self), PyObject *args)
{
    // ...省略...
    result = (double) ((arg1)->x); // <- 整个函数的目的就是封装这一句
    // ...省略...
}
```

和成员变量相似，对于类的成员函数，SWIG 也会进行类似的封装。例如对于下面的 CPoint 类的成员函数 power()：

```
class CPoint
{
```

```
public:
    double x, y;
    double power();
};
```

在 Python 中可以如下调用：

```
>>> import demo
>>> p = demo.CPoint()
>>> p.x = 3
>>> p.y = 4
>>> p.power()
25.0
```

查看"demo.py"中的源代码，可以找到：

```
class CPoint(_object):
    # ...省略...
    def power(self): return _demo.CPoint_power(self)
```

而在"demo_wrap.cpp"中，可以找到 CPoint_power()对应的封装函数：

```
SWIGINTERN PyObject *_wrap_CPoint_power(PyObject *SWIGUNUSEDPARM(self), PyObject *args)
{
    // ...省略...
    result = (double)(arg1)->power();
    // ...省略...
}
```

4. 类型映射

SWIG 的主要任务就是自动生成一个封装函数，在 C/C++函数参数和 Python 函数参数之间进行相互转换。它的所有转换规则都可以使用类型映射(Typemaps)进行定义。SWIG 已经提供了许多默认的类型映射定义。因此在前面的例子中，我们完全没有意识到它的存在。但有时为了让封装函数更接近 Python 的习惯，需要使用类型映射对封装函数的参数进行修改。

例如，C 语言的函数无法返回多个值，因此通常的做法是采用指针实现这一目的：

```
void add_mult(double x, double y, double * s, double *p)
{
    *s = x + y;
    *p = x * y;
}
```

如果直接使用 SWIG 对 add_mult()进行封装，将无法通过函数的声明分析出指针 s 和 p 只

是为了返回值，因此最终的 Python 函数也需要有 4 个参数，并且后两个参数是指针类型的对象，这样的函数在 Python 中很不好用。我们可以通过类型映射改变 SWIG 的默认行为，如果告诉 SWIG 指针 s 和 p 只是为了返回值，那么 SWIG 产生的 Python 函数只需要两个参数，并且将两个返回值用列表返回。类型映射用起来很简单，只需要在接口文件“demo.i”中对 add_mult()进行如下声明：

```
void add_mult(double x, double y, double * OUTPUT, double *OUTPUT);
```

其中，“double *OUTPUT”不但是函数的参数说明，它还是一个已经定义的类型映射，它告诉 SWIG 这个 double 指针参数是用来输出值的。如果希望在函数声明中保留原来的参数名，可以使用“%apply”命令将类型映射运用到指定名称的参数之上。下面的定义先将类型映射“double *OUTPUT”运用到参数“double *s”和“double *p”之上。由于“%apply”的作用会一直有效，因此如果希望结束对指针 s 和 p 的类型映射，就需要使用“%clear”进行清除：

```
%apply double *OUTPUT { double *s, double *p };
void add_mult(double x, double y, double * s, double *p);
%clear double *s, double *p;
```

这样一来，在 Python 中 add_mult()的调用就变得很直观了：

```
>>> import demo
>>> demo.add_mult(3,4)
[7.0, 12.0]
```

除了 OUTPUT 之外，还有 INPUT 和 INOUT 等常用的类型映射。INPUT 将 C 语言的指针变量转换为 Python 中它所指向的类型。例如下面的 add()要求其两个参数为指向 double 类型的指针，默认情况下，在 Python 中它是用 SwigPyObject 类型的指针对象来表示。但是我们希望 add()能直接接受浮点数值进行运算，因此可以使用 INPUT 类型映射：

```
double add(double *INPUT, double *INPUT);
```

在 Python 中可以直接将能转换为浮点数的对象传递给 add()，而不再需要用指针对象：

```
>>> demo.add(4,5)
9.0
```

有时候，C 语言中指针类型的参数同时起到输入和输出的作用，例如下面的 inc()，它将其参数指针指向的变量值增加 1：

```
void inc(int *INOUT);
```

由于 Python 中的整数对象是不可变的，因此 Python 中的 inc()无法直接更改其参数对象的值。这时可以使用 INOUT 类型映射，它告诉 SWIG 此参数的值在函数中参与运算，并且它的

值会被更新，以返回值的形式返回新的值：

```
>>> demo.inc(4)
5
```

如果希望用它来更新某个变量的值，只能将同一变量重新绑定为 inc()所返回的值：

```
>>> x = 2
>>> x = demo.inc(x)
>>> x
3
```

5. 函数指针

在 C 语言中，可以使用函数指针将函数作为参数传递；在 SWIG 中，可以将 C 语言的函数封装成一个 Python 的指针对象，并在 Python 中将它传递给别的 C 语言函数，实现函数指针的传递。

例如，在下面的 C 语言程序中，sum_func()的第一个参数是函数指针，此外还定义有 square()、reciprocal()和 linear()三个函数，它们可以作为函数指针传递给 sum_func()：

```
double sum_func(double (*op)(double), int s, int e)
{
    double sum = 0;
    int i;
    for(i=s;i<e;i++)
    {
        sum += (*op)(i);
    }
    return sum;
}

double square(double x)
{
    return x*x;
}

double reciprocal(double x)
{
    return 1/x;
}

double linear(double x)
{
    return 0.5*x+1;
}
```

如果直接将 square()等三个函数声明放到接口文件中，就会生成三个封装函数，而不是它们的指针。为了让 SWIG 同时生成函数指针对象和封装函数，可以使用“%callback”命令：

```
double sum_func(double (*op)(double), int s, int e);

%callback("cb_%s");
double square(double x);
double reciprocal(double x);
double linear(double x);
%nocallback;
```

对于在“%callback”和“%nocallback”中包围的函数声明，将同时生成其封装函数和函数指针对象。为了解决二者的命名冲突问题，通过“%callback”命令的参数“cb_%s”，可以指定函数指针对象名的格式化字符串。因此对于 square()，它所对应的指针对象就是 cb_square：

```
>>> import demo
>>> demo.reciprocal(10) # 直接调用封装函数
0.10000000000000001
>>> demo.reciprocal # 类型为函数
<built-in function reciprocal>
>>> demo.sum_func(demo.cb_reciprocal, 1, 100) # 用函数指针对象将函数作为参数传递
5.1773775176396208
>>> import numpy as np
>>> np.sum( 1.0/np.arange(1,100)) # 验证结果
5.1773775176396208
>>> demo.cb_reciprocal # 类型为指针对象
<Swig Object of type 'double (*)(double)' at 0x00E3F818>
```

6. 回调 Python 函数

使用函数指针，只能将 C/C++语言的函数经过 Python 中转之后再传递给 C/C++语言的函数，无法实现 C/C++语言程序调用 Python 函数。前面介绍过，C++的类可以被封装成 Python 的类，并且在 Python 中还可以继承此封装类。但是默认情况下，C++的类无法获知 Python 类的继承情况，因此在 C++类中定义的虚函数，无法通过其多态性调用 Python 子类中的实现函数。为了实现 C++调用 Python，我们需要开启“Director”功能——为接口文件的第一行“%module”命令添加参数 directors="1"：

```
%module(directors="1") demo
```

然后，对于希望调用 Python 函数的 C++类 Foo，通过“%feature”命令开启它的“Director”功能：

```
%feature("director") Foo;
```

下面仍然以函数值求和为例，介绍如何实现 C++调用 Python。在文件“demo.h”中，计算求和的 C++类的源程序如下，其中 Func()是一个虚函数，我们希望在 Python 的子类中通过覆盖它实现不同数学函数的求和计算：

```
class Sum
{
    public:
    double Cal(int s, int e)
    {
        double sum = 0;
        for(int i=s;i<e;i++)
            sum += Func(i);
        return sum;
    }
    virtual double Func(double x){return x;}
};
```

通过修改“demo.i”的第一行，开启“Director”功能，并添加如下的 Sum 类的声明：

```
%feature("director") Sum;
class Sum
{
    public:
    double Cal(int s, int e);
    virtual double Func(double x);
};
```

编译成扩展模块之后，在 Python 中可以按如下方式使用 Sum 类：

```
>>> import demo
>>> demo.Sum().Cal(1,101) # 直接使用 Sum 类，计算 1 到 100 的和
5050.0
>>> class SumReciprocal(demo.Sum): # 定义 Sum 的子类
...     def Func(self, x):  # 覆盖其 Func()虚函数
...         return 1.0/x
...
>>> SumReciprocal().Cal(1, 100) # 计算 1 到 99 的倒数的和
5.1773775176396208
```

16.4.3 SWIG 处理 NumPy 数组

用 C 语言编写扩展模块的最重要目的就是提高数组运算的效率，因此我们希望 SWIG 能够方便地处理 NumPy 数组。标准的 SWIG 并不支持 NumPy 数组，需要下载一个 NumPy 的接

口文件“numpy.i”，读者也可以在本节的示例程序文件夹下找到它。



http://projects.scipy.org/numpy/browser/trunk/doc/swig
从此链接下载 NumPy 的 SWIG 接口文件“numpy.i”



swig_numpy\demo.cpp, demo.h, demo.i, setup.py
用 SWIG 编写处理 NumPy 数组的扩展模块

为了在扩展模块中处理 NumPy 数组，接口文件“demo.i”中必须有如下内容：

```
%module demo

%{
    #define SWIG_FILE_WITH_INIT
    #include "demo.h"
%}

%include "numpy.i"

%init %{
    import_array();
%}
```

用“%init”命令括起来的内容，将输出到扩展模块的初始化函数 SWIG_init()中。而为了在其中调用 import_array()，还需要定义：

```
#define SWIG_FILE_WITH_INIT
```

由于扩展模块使用了 NumPy，它需要引用 NumPy 的 C 语言的头文件，因此在文件“setup.py”中指定 NumPy 的头文件的位置：

```
demo_module = Extension(
    '_demo',
    ['demo.i', 'demo.cpp'],
    include_dirs = [numpy.get_include()],
    swig_opts=["-c++"]
)
```

按照上面的步骤设置好接口文件“demo.i”和“setup.py”之后，就可以正常编译处理 NumPy 数组的扩展模块了。

在“numpy.i”中定义了许多类型映射以实现各种参数转换要求。例如对于下面的 drange()，我们希望在 Python 中它和 arange()的功能一样：

```
void drange(double * x, int n)
{
    for(int i=0;i<n;i++) x[i]  = i;
}
```

因为 drange()的数组参数 x 只作为输出数据用，所以可以使用类型映射将此函数改为返回一个数组，下面是接口文件中 drange()的声明：

```
void drange(double * ARGOUT_ARRAY1, int DIM1);
```

扩展模块中 drange()的调用方法如下：

```
>>> import demo
>>> demo.drange(10)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```


所返回的数组的内存分配工作由“demo_wrap.cpp”中的封装函数完成。这里用到的类型映射为(DATA_TYPE* ARGOUT_ARRAY1, DIM_TYPE DIM1)，类型映射可以对连续的多个参数进行类型转换。ARGOUT_ARRAY1 表示此参数数组将作为函数返回值，并且它是一维的。读者可以在文件“numpy.i”中找到它的定义，并且可以在此文件中找到它所支持的所有类型映射的说明[⑥]。

例如，可以找到一个返回固定大小的二维数组用的类型映射——ARGOUT_ARRAY2[ANY][ANY]。下面的例子用它返回二维空间中的旋转矩阵：

```
void rot2d(double x[3][3], double th)
{
    x[0][2] = 0; x[1][2] = 0; x[2][0] = 0; x[2][1] = 0;
    x[0][0] = cos(th); x[0][1] = -sin(th); x[1][0] = sin(th); x[1][1] = cos(th);
    x[2][2] = 1;
}
```

在接口文件中使用如下的定义：

```
void rot2d(double ARGOUT_ARRAY2[3][3], double th);
```

在 Python 中，rot2d()的使用方法为：

```
>>> demo.rot2d(np.pi/4)
array([[ 0.70710678, -0.70710678,  0.        ],
       [ 0.70710678,  0.70710678,  0.        ],
       [ 0.        ,  0.        ,  1.        ]])
```

⑥ 在“numpy.i”中搜索“%numpy_typemaps() macro”可以快速找到它。

在“numpy.i”中找不到返回二维数组的内存映射(DATA_TYPE* ARGOUT_ARRAY2, DIM_TYPE DIM1, DIM_TYPE DIM2)。因此如果需要返回可指定大小的二维数组，只能先如drange()一样得到一个一维数组，然后在 Python 中将其形状修改为二维，或者使用后面将要介绍的 INPLACE_ARRAY2 类型映射。

用 IN_ARRAY 相关的类型映射表示参数只作为输入数据使用。由于在封装函数中会对参数进行类型转换，因此在 Python 中调用时，可以给它传递任意的序列对象，只要此序列对象的元素都能够正确地转换为函数所需的数据类型即可。例如，对于下面的求平方和函数：

```
double sum_power(double * x, int n)
{
    double sum = 0;
    for(int i=0;i<n;i++) sum += x[i]*x[i];
    return sum;
}
```

在接口文件中使用下面的定义：

```
double sum_power(double * IN_ARRAY1, int DIM1);
```

在 Python 中，可以如下使用 sum_power()：

```
>>> demo.sum_power([1,2,3]) # 可以对列表进行计算
14.0
>>> x = np.arange(10.0)
>>> demo.sum_power(x[::2]) # 数组的元素可以不是连续的
120.0
```

显然用 IN_ARRAY 进行类型映射时，在封装函数内部会产生一个临时数组用来保存输入序列中的每个元素，而实际的 C 语言函数则对这个临时数组中的元素进行计算。

用 INPLACE_ARRAY 表示数组的内容将会被 C 语言程序更新，因此 Python 所传递的参数必须是 NumPy 数组，其元素类型和 C 语言的对应参数类型必须一致，并且数组元素必须占用连续的内存空间。例如下面的 power()计算数组中每个元素的平方：

```
void power(double *x, int n)
{
    for(int i=0;i<n;i++) x[i] *= x[i];
}
```

其接口定义为：

```
void power(double *INPLACE_ARRAY1, int DIM1);
```

在 Python 中：

```
>>> a = np.arange(10.0)
>>> demo.power(a)
>>> a
array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,  81.])
>>> demo.power([1,2,3]) # 参数不能是列表
[[省略]]
TypeError: Array of type 'double' required.  A 'list' was given
>>> demo.power(a[::2]) # 元素位置不连续的数组也不行
[[省略]]
TypeError: Array must be contiguous.  A non-contiguous array was given
```

可以看出使用 INPLACE_ARRAY 时，C 语言函数所处理的就是 NumPy 数组的数据存储区，由于我们没有将 NumPy 数组的 strides 属性传递给 C 语言函数，因此它只能处理连续存储的数组。

第 17 章

自适应滤波器

近年来，随着数字信号处理器功能的不断增强，自适应信号处理(adaptive signal process)经常用于噪声消除、回声控制、信号预测、声音定位等众多的信号处理领域。

本章简要介绍自适应滤波器的原理及其最常用的算法 NLMS，并使用纯 Python 实现 NLMS 算法，在此基础上我们对自适应滤波器的各种应用做了一些模拟计算。最后为了提高运算速度，我们用 SWIG 和 Weave 对 C 语言的 NLMS 程序进行封装。

17.1 自适应滤波器简介

对于许多数字信号处理方面的应用来说，由于事先并不知道系统的一些参数，例如噪声的特性、未知系统的传递函数等等，因此经常需要滤波器能够根据输入信号自动调节其参数以进行数字滤波。

随着数字信号处理芯片(DSP)的性能不断增强，自适应滤波器已经广泛应用于通信、电子、交通、医疗等各个领域。尽管其应用领域十分广泛，但基本的系统构造大致可分为系统识别、信号预测、信号均衡等几种。

17.1.1 系统识别

所谓系统识别(system identification)，是指通过对未知系统的输入和输出信号进行观测，构造一个滤波器使得它在相同输入信号的情况下，输出信号和未知系统相同。简而言之，就是通过观测未知系统对输入信号的反应，探知其内部情况。为了探知内情而使用的输入信号，我们称之为参照信号。

如图 17-1 所示，参照信号 $x(j)$同时输入到未知系统和自适应滤波器 H 中，未知系统的输出为 $y(j)$， 自适应滤波器的输出为 $u(j)$。由于观测误差或受外部噪声的干扰，实际观测到的未知系统的输出为 $d(j)=y(j)+n(j)$，$n(j)$被称为外部干扰。$d(j)$和 $u(j)$之间的误差 $e(j)=d(j)-u(j)$，也就是自适应滤波器 H 的输出和未知系统的输出之间的误差，通过这个误差我们更新 H 的内部系数，使得它的输出更加靠近未知系统的输出。

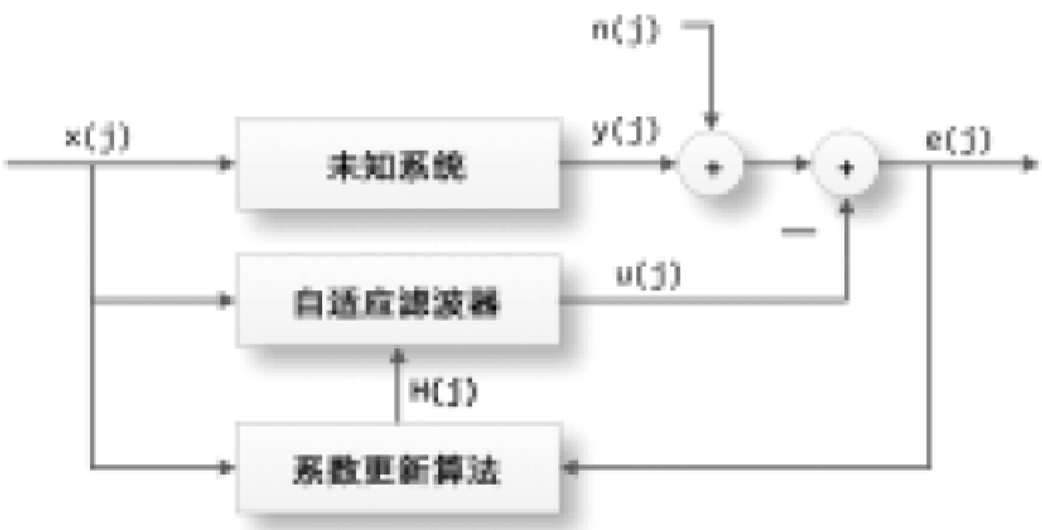


图 17-1　系统识别的系统框图

上面各个公式中的 j 表示某一时刻，由于我们讨论的是数字信号处理，已经对所有的信号进行了取样，因此可以把 j 简单地看做取样后信号数据的下标。

17.1.2　信号预测

所谓信号预测，就是通过信号过去的值计算现在的值，图 17-2 是信号预测的系统框图。

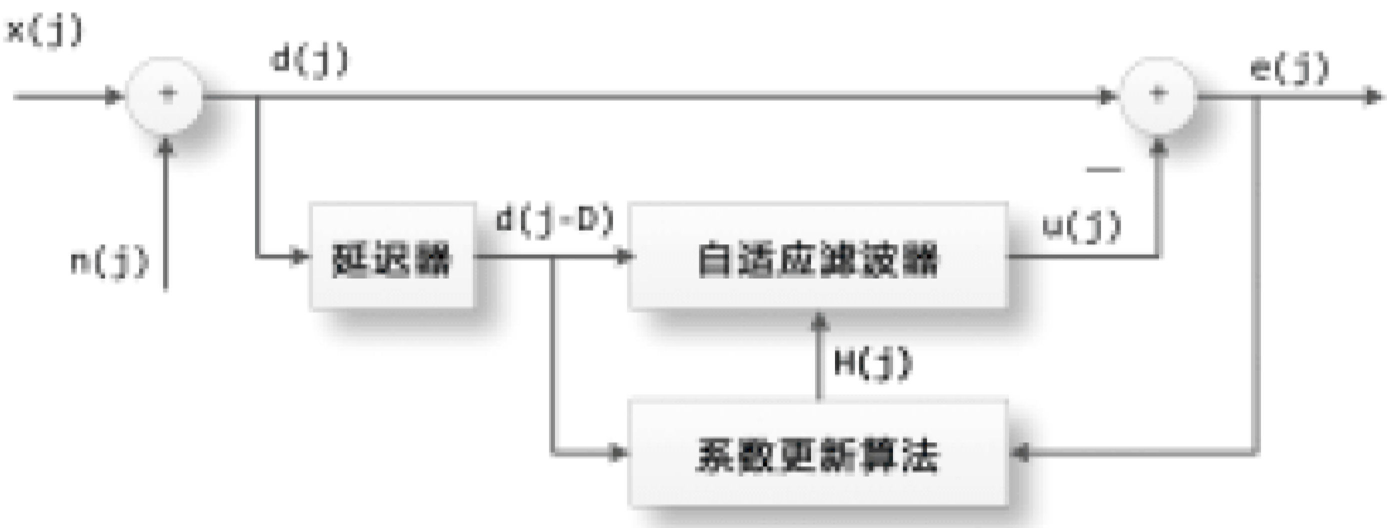


图 17-2　信号预测的系统框图

x(j)是待测信号，假设我们无法完美地观测此信号，因此导入一个外部干扰 n(j)。这样一来，d(j)=x(j)+n(j)就是我们观测到的待测信号。

通过延迟器将 d(j)进行延时得到 d(j−D)，并把 d(j−D)输入到自适应滤波器 H 中，得到其输出为 u(j)，u(j)就是自适应滤波器通过待预测信号过去的值预测到的现在的值，计算观测值 d(j)和预测值 u(j)之间的误差 e(j)=d(j)−u(j)，通过 e(j)更新自适应滤波器 H 的内部系数，使得其输出更加接近 d(j)。

如果 x(j)存在白噪声的成分和周期信号的成分，那么由于白噪声是完全随机、无法预测的信号，因此用过去的值 x(j−D)所能预测的只能是其中的周期信号的成分。这样一来，自适应滤波器 H 的输出信号 u(j)就会与周期信号成分渐渐逼近，而 e(j)则是剩下的不可预测的白噪声的成分。因此自适应滤波器也可以应用于噪声消除。

17.1.3　信号均衡

如图 17-3 所示，当信号 x(j)通过未知系统后变成了 y(j)，未知系统对信号 x(j)进行了

某种改变，使得其波形产生歪曲。我们希望通过均衡器矫正这种歪曲，也就是通过 y(j)重建原始信号 x(j)，由于因果律，还原原始信号 x(j)是不可能的，我们只能还原其延时了的信号 x(j−D)。x(j)和 x(j−D)除了时间上的延迟之外，其他特性完全相同。

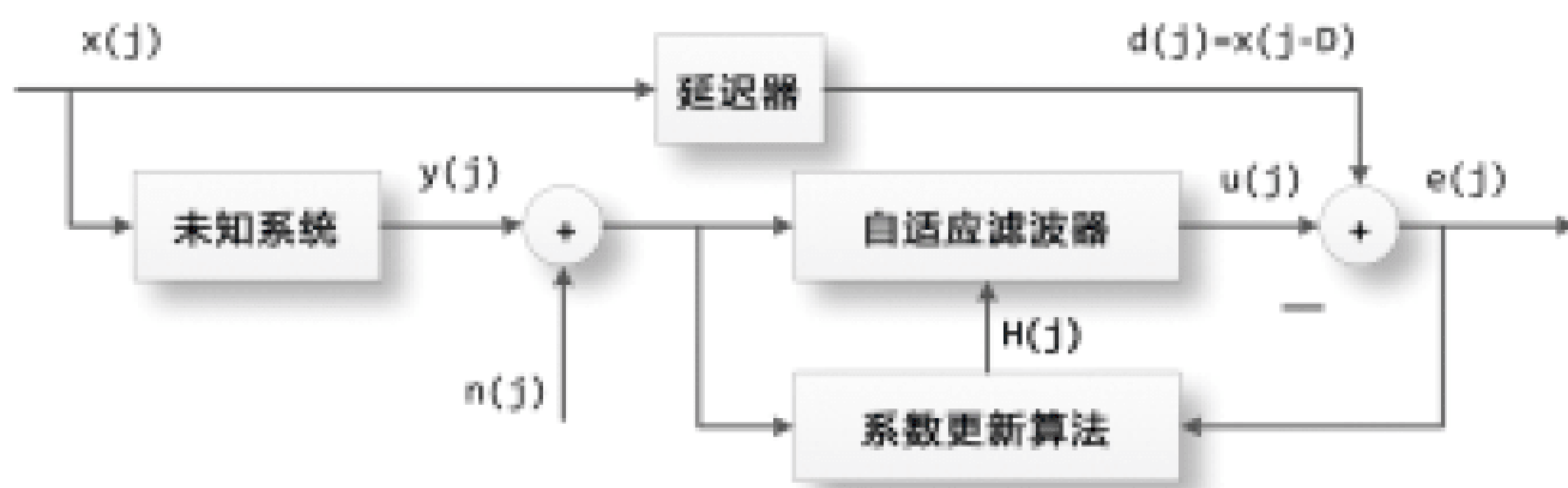


图 17-3　信号均衡的系统框图

这里我们将观测到的未知系统的输出 y(j)+n(j)输入到自适应滤波器 H 中，通过 H 的系数更新使得其输出 u(j)逐渐逼近原始信号的延时 x(j−D)。这样一来，我们就构建了一个滤波器 H，使得它与未知系统的卷积正好等于一个脉冲传递函数。也就是说，H 的频域特性恰好能抵消未知系统所带来的改变。

17.2　NLMS 计算公式

自适应滤波器中最重要的一个环节就是其系数的更新算法，如果不对自适应滤波器的系数更新，那么它就只是一个普通的滤波器了。系数更新算法有很多种，最简单的一种方法叫做 NLMS(归一化最小均方)，让我们先看看它的数学公式表达。

设置自适应滤波器系数 h 的所有初始值为 0，假设 h 的长度为 I。

$$h(0)=0$$

对每个取样值进行如下计算，其中 n = 0, 1, 2, …

$$x(n)=[x(n),x(n-1),...,x(n-I+1)]^T$$

$$e(n)=d(n)-h^T(n)x(n)$$

$$h(n+1)=h(n)+\frac{\mu e(n)x(n)}{x^T(n)x(n)}$$

自适应滤波器系数 h 是一个长度为 I 的矢量，也就是一个长度为 I 的 FIR 滤波器。在时刻 n，滤波器对应的输入信号为 $x(n)$，它也是一个长度为 I 的矢量。这两个矢量的点乘即为滤波器的输出。它和目标信号 $d(n)$之间的差为 $e(n)$，然后根据 $e(n)$和 $x(n)$，更新滤波器的系数。

数学公式总是令人难以理解的，下面我们以图 17-4 为例进行说明。

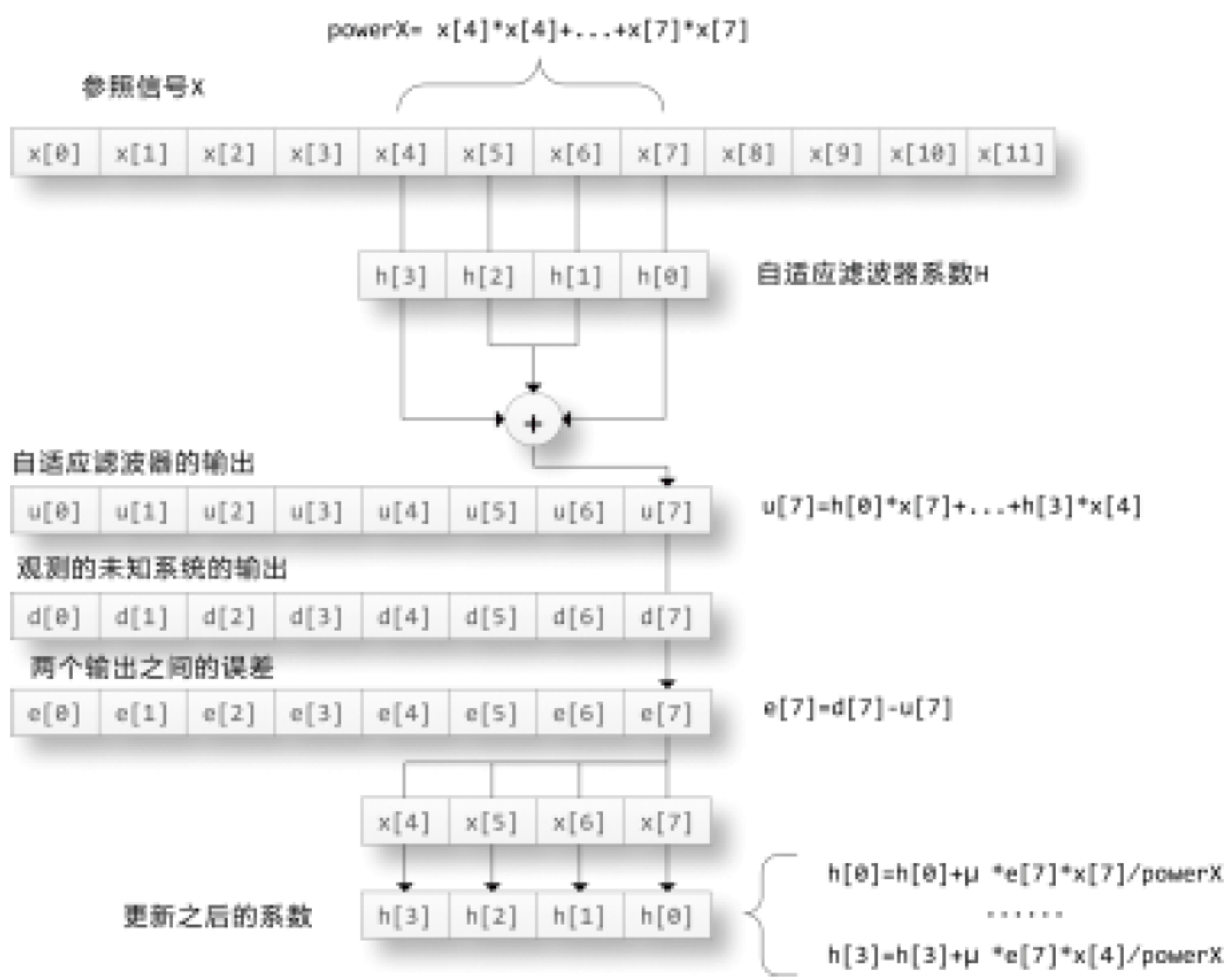


图 17-4　NLMS 算法示意图

图 17-4 中假设自适应滤波器 H 的长度为 4，在时刻 7 滤波器的输出为：

```
u[7] = h[0]*x[7] + h[1]*x[6] + h[2]*x[5] + h[3]*x[4]
```

滤波器的输入信号的平方和 powerX 为：

```
powerX = x[4]*x[4] + x[5]*x[5] + x[6]*x[6] + x[7]*x[7]
```

未知系统的输出 d[7]和滤波器的输出 u[7]之间的差为：

```
e[7] = d[7] - u[7]
```

使用 e[7]和 x[4]到 x[7]对滤波器的系数进行更新：

```
h[3] = h[3] + u * e[7]*x[4]/powerX
h[2] = h[2] + u * e[7]*x[5]/powerX
h[1] = h[1] + u * e[7]*x[6]/powerX
h[0] = h[0] + u * e[7]*x[7]/powerX
```

对于每个取样值都需要进行上述计算。参数 u 为更新系数，取值范围一般在 0~1 之间。值越大，系数更新的速度越快。

17.3 用 NumPy 实现 NLMS 算法

按照刚才介绍的 NLMS 算法，很容易写出用 NumPy 实现的 NLMS 程序：



nlms_numpy.py
用 NumPy 实现 NLMS 算法

```
import numpy as np

# 用 NumPy 实现 NLMS 算法
# x 为参照信号，d 为目标信号，h 为自适应滤波器的初值
# step_size 为更新系数
# 返回自适应滤波器的输出信号
def nlms(x, d, h, step_size=0.5):
    count = min(len(x), len(d)) ❶
    u = np.zeros(count, dtype=np.float64)

    nh = len(h)
    # 计算输入到 h 中的参照信号的平方和
    power = np.sum( x[:nh] **2 )  ❷

    i = nh ❸
    while True:
        x_input = x[i:i-nh:-1] ❹
        u[i] = np.dot(x_input , h)
        e = d[i] - u[i]
        h += step_size * e / power * x_input

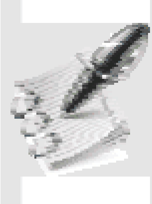
        # 减去最早的取样
        power -= x_input[-1] * x_input[-1]
        i+=1
        if i >= count: return u
        # 增加最新的取样
        power += x[i] * x[i]
```

nlms()的输入为参照信号 x、目标信号 d 和自适应滤波器的初始系数 h，返回自适应滤波器的输出信号 u，并且更新系数 h。

❶首先取 x 和 d 中较小的长度作为循环的次数 count，并且用它初始化保存滤波器输出信号的数组 u。❷为了节省计算时间，我们用一个临时变量 power 保存输入到滤波器 h 中的参照信号 x 的能量。在对 x 中的每个取样值进行循环时，只需要从 power 中减去 x 中最早的一个取样值的平方，加上最新的取样值的平方，就能保证 power 始终等于输入信号的平方和。这样一

来，每次循环只需要计算两次乘法和两次加法即可。❸循环变量 i 的初始值本来可以从 nh − 1 开始，但是为了让❹处的 i − nh 不为负，我们让循环变量从 nh 开始。

为了对自适应滤波器的各种应用进行模拟，我们还需要如下的几个辅助函数。



nlms_common.py
自适应滤波器模拟的辅助函数库

```
import numpy as np
import pylab as pl

# 随机产生 FIR 滤波器的系数，长度为 length，延时为 delay，指数衰减
def make_path(delay, length):
   plen = length - delay
   h = np.zeros(length, np.float64)
   h[delay:] = np.random.standard_normal(plen) * np.exp( np.linspace(0, -4, plen) )
   h /= np.sqrt(np.sum(h*h))
   return h

def plot_converge(y, u, label=""):
    size = len(u)
    avg_number = 200
    e = np.power(y[:size] - u, 2)
    tmp = e[:int(size/avg_number)*avg_number]
    tmp.shape = -1, avg_number
    avg = np.average( tmp, axis=1 )
   pl.plot(np.linspace(0, size, len(avg)), 10*np.log10(avg), linewidth=2.0, label=label)

def diff_db(h0, h):
   return 10*np.log10(np.sum((h0-h)*(h0-h)) / np.sum(h0*h0))
```

make_path()产生一个长度为 length、最小延时为 delay 的指数衰减的波形。这种波形和封闭空间的声音的传递函数比较近似，因此在计算机上进行声音的自适应滤波器算法模拟时经常用这种波形作为系统的传递函数。

plot_converge()绘制信号 y 和 u 之间的误差，每 avg_number 个取样点就计算一次两个信号之间误差的平方的平均值。我们用它绘制未知系统的输出 y 和自适应滤波器的输出 u 之间的误差。观察自适应滤波器是如何收敛的，以评价自适应滤波器的收敛特性。

diff_db()同样用来评价自适应滤波器的收敛特性，不过它是直接计算未知系统的传递函数 h0 和自适应滤波器的传递函数 h 之间的误差。下面我们会看到，这两个函数得到的收敛值是相同的。

17.3.1 系统辨识模拟

我们用下面的函数调用 nlms()，对图 17-1 所示的系统识别进行模拟：



system_identify.py
使用 NLMS 对系统识别进行模拟

```
# 用 NLMS 对系统识别进行模拟，未知系统的传递函数为 h0，使用的参照信号为 x
def sim_system_identify(nlms_func, x, h0, step_size, noise_scale):
      y = np.convolve(x, h0)
      d = y + np.random.standard_normal(len(y)) * noise_scale # 添加白噪声的外部干扰
      h = np.zeros(len(h0), np.float64) # 自适应滤波器的长度和未知系统相同，初始值为 0
      u = nlms_func( x, d, h, step_size )
      return y, u, h
```

其中：nlms_func 参数为 NLMS 算法的实现函数；x 参数为参照信号；h0 参数为未知系统的传递函数；step_size 参数为 NLMS 算法的更新系数；noise_scale 参数为外部干扰的系数，它决定外部干扰的强弱，0 表示没有外部干扰。

在函数的返回值中，y 是不包括外部干扰的未知系统的输出；u 是自适应滤波器的输出；h 是自适应滤波器最终的系数。

我们用下面的函数创建未知系统 h0 和参照信号 x，然后调用 sim_system_identify()得到结果并且绘图：

```
def system_identify_test1():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(10000)  # 参照信号为白噪声
    y, u, h = sim_system_identify(nlms, x, h0, 0.5, 0.1)
    print diff_db(h0, h)
    pl.figure( figsize=(8, 6) )
    pl.subplot(211)
    pl.subplots_adjust(hspace=0.4)
    pl.plot(h0, c="r")
    pl.plot(h, c="b")
    #pl.title(u"未知系统和收敛后的滤波器的系数比较")
    pl.subplot(212)
    plot_converge(y, u)
    #pl.title(u"自适应滤波器收敛特性")
    pl.xlabel(u"迭代次数 (取样点)")
    pl.ylabel(u"收敛程度(dB)")
```

图 17-5(上)显示的是未知系统(红色)和自适应滤波器(蓝色)的传递函数的系数，可以看到，自适应滤波器已经十分接近未知系统了。diff_db(h0, h)的输出为-25.35dB。图 17-5(下)显示 y 和

u 之间的误差，可以从中观察到自适应滤波器的收敛过程。我们看到，经过约 3000 点的计算之后，收敛过程已经饱和，最终的误差为-25dB 左右，和 diff_db()的计算结果一致。

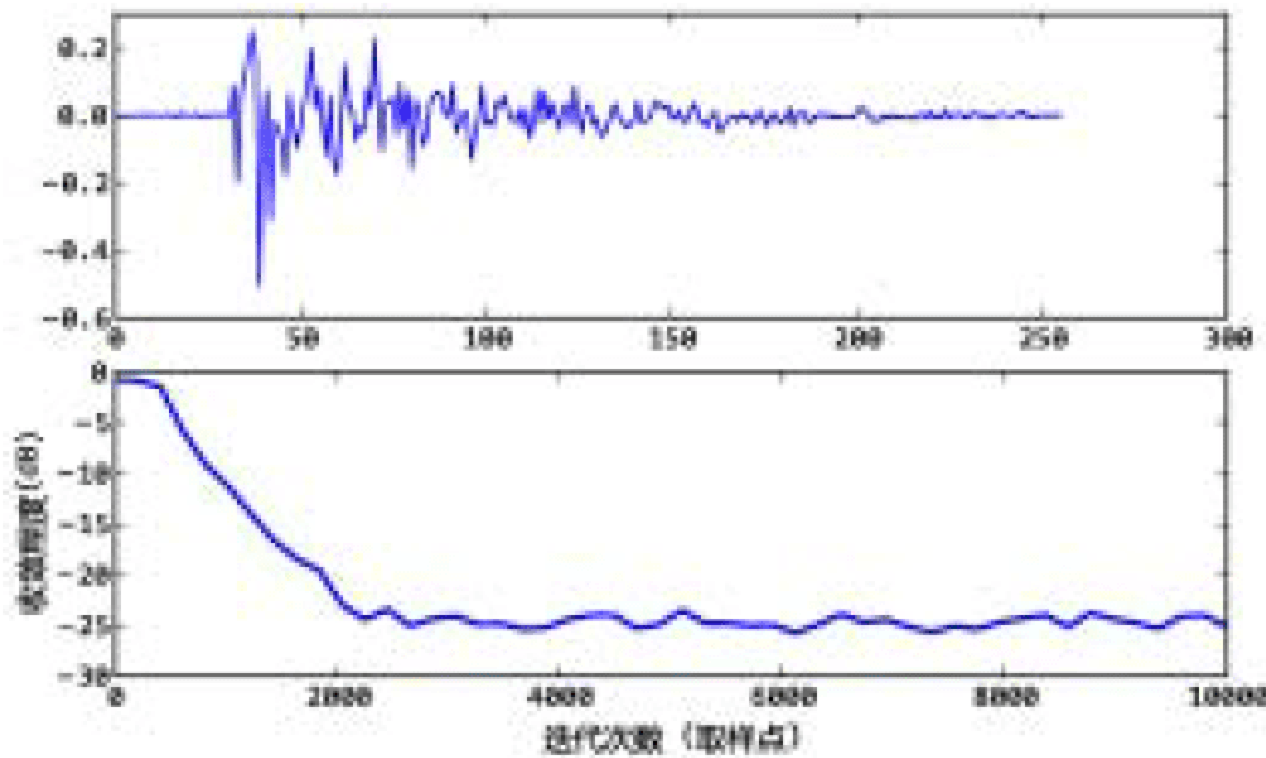


图 17-5 自适应滤波器收敛之后的系数(上图)和收敛速度(下图)

从图 17-5 可以看到收敛过程的两个重要特性：收敛时间和收敛精度。参照信号的特性、外部干扰的强弱和更新系数都会影响这两个特性。下面让我们看看参照信号为白噪声、外部干扰的能量固定时，更新系数对它们影响，结果如图 17-6 所示(见文前彩插)。由图可知：更新系数越小，收敛速度越慢，但收敛精度越高。

```
def system_identify_test2():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for step_size in np.arange(0.1, 1.0, 0.2):
        y, u, h = sim_system_identify(nlms, x, h0, step_size, 0.1)
        plot_converge(y, u, label=u"μ=%s" % step_size)
    #pl.title(u"更新系数和收敛特性的关系")
    pl.xlabel(u"迭代次数 (取样点)")
    pl.ylabel(u"收敛程度(dB)")
    pl.legend()
```

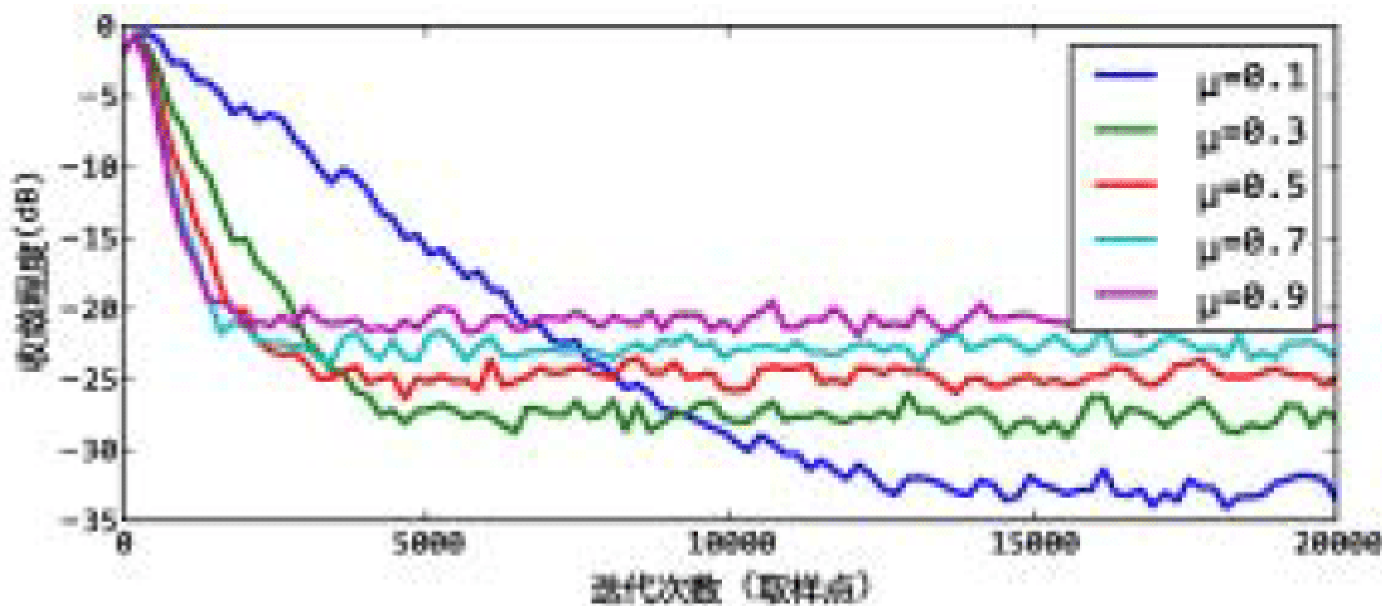


图 17-6　更新系数和收敛速度的关系

下面的语句用来计算外部干扰能量变化时的收敛特性：

```
def system_identify_test3():
    h0 = make_path(32, 256) # 随机产生一个未知系统的传递函数
    x = np.random.standard_normal(20000)  # 参照信号为白噪声
    pl.figure(figsize=(8,4))
    for noise_scale in [0.05, 0.1, 0.2, 0.4, 0.8]:
        y, u, h = sim_system_identify(nlms, x, h0, 0.5, noise_scale)
        plot_converge(y, u, label=u"noise=%s" % noise_scale)
    #pl.title(u"外部干扰和收敛特性的关系")
    pl.xlabel(u"迭代次数 (取样点)")
    pl.ylabel(u"收敛程度(dB)")
    pl.legend()
```

从图 17-7 可以看出，当外部干扰的振幅增加一倍、能量增加 6 dB 时，收敛精度降低 6 dB(见文前彩插)。而由于更新系数相同，因此收敛过程中的收敛速度是一样的。

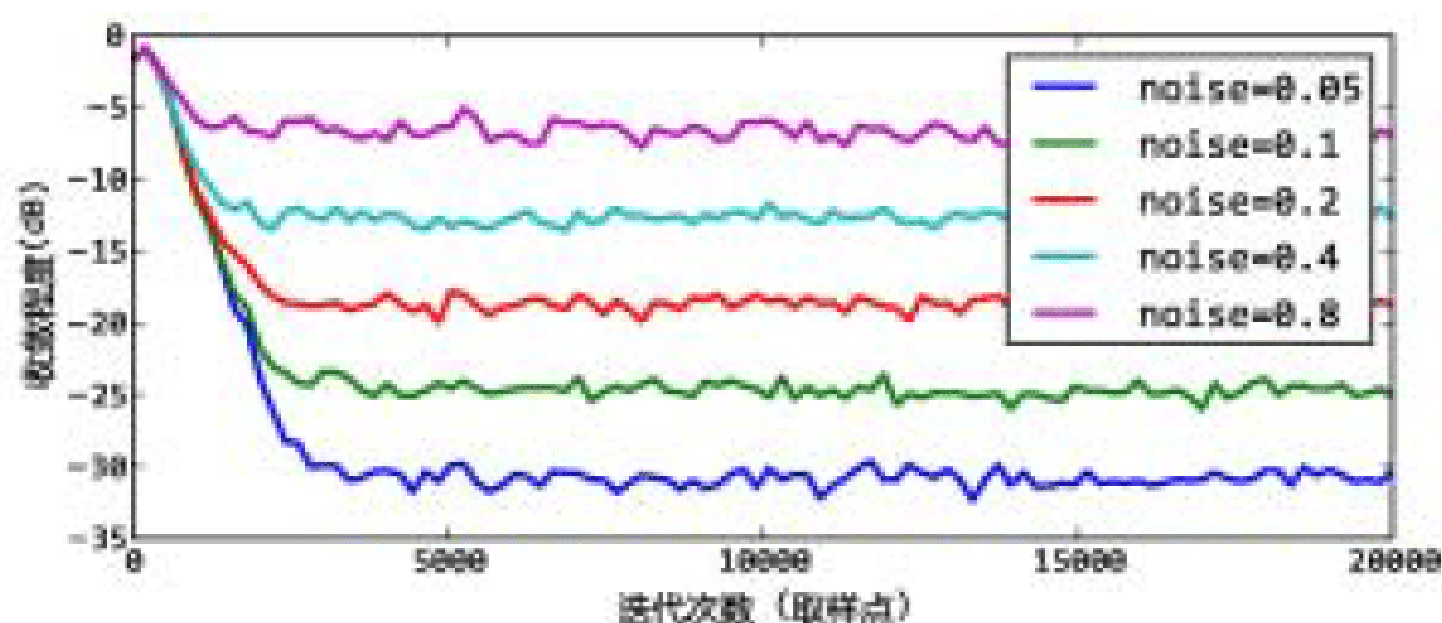


图 17-7　外部干扰噪声和收敛速度的关系

17.3.2　信号均衡模拟

对于图 17-3 所示的信号均衡系统，我们用如下的程序进行模拟：



signal_equalization.py
自适应信号均衡模拟

```
def sim_signal_equalization(nlms_func, x, h0, D, step_size, noise_scale):
    d = x[:-D] ❶
    x = x[D:]
    y = np.convolve(x, h0)[:len(x)] ❷
    h = np.zeros(2*len(h0)+2*D, np.float64) ❸
    y += np.random.standard_normal(len(y)) * noise_scale
    u = nlms_func(y, d, h, step_size)
    return h
```

sim_signal_equation()的参数与前面的 sim_system_identify()类似，增加了一个 D 参数，用于表示延迟器的延时。

在函数中，❶首先对输入信号进行延时，❷然后计算未知系统的输出信号，❸并初始化一个足够长的数组 h 作为自适应滤波器的系数。这里数组 h 的长度为两倍延时加两倍未知系统的传递函数的长度。

函数的返回值为自适应滤波器收敛后的系数，它能够均衡 h0 对输入信号所造成的影响。我们通过下面的程序产生数据、调用模拟函数以及绘制结果：

```
def signal_equalization_test1():
    import scipy.signal
    h0 = make_path(5, 64)
    D = 128
    length = 20000
    data = np.random.standard_normal(length+D)
    h = sim_signal_equalization(nlms, data, h0, D, 0.5, 0.1)
    pl.figure(figsize=(8,4))
    pl.plot(h0, label=u"未知系统")
    pl.plot(h, label=u"自适应滤波器")
    pl.plot(np.convolve(h0, h), label=u"二者卷积")
    #pl.title(u"信号均衡演示")
    pl.legend()
    w0, H0 = scipy.signal.freqz(h0, worN = 1000)
    w, H = scipy.signal.freqz(h, worN = 1000)
    pl.figure(figsize=(8,4))
    pl.plot(w0, 20*np.log10(np.abs(H0)), w, 20*np.log10(np.abs(H)))
    #pl.title(u"未知系统和自适应滤波器的振幅特性")
    pl.xlabel(u"圆频率")
    pl.ylabel(u"振幅(dB)")
```

如果延迟器的延时 D 不够，那么会由于因果律，使得自适应滤波器无法收敛。因此这里我们采用的延时为 h0 长度的两倍。图 17-8 显示了 h0、h 以及它们的卷积(见文前彩插)。我们看到，h0 和 h 的卷积正好是一个脉冲信号，脉冲发生的时刻正好等于指定的延时 128。

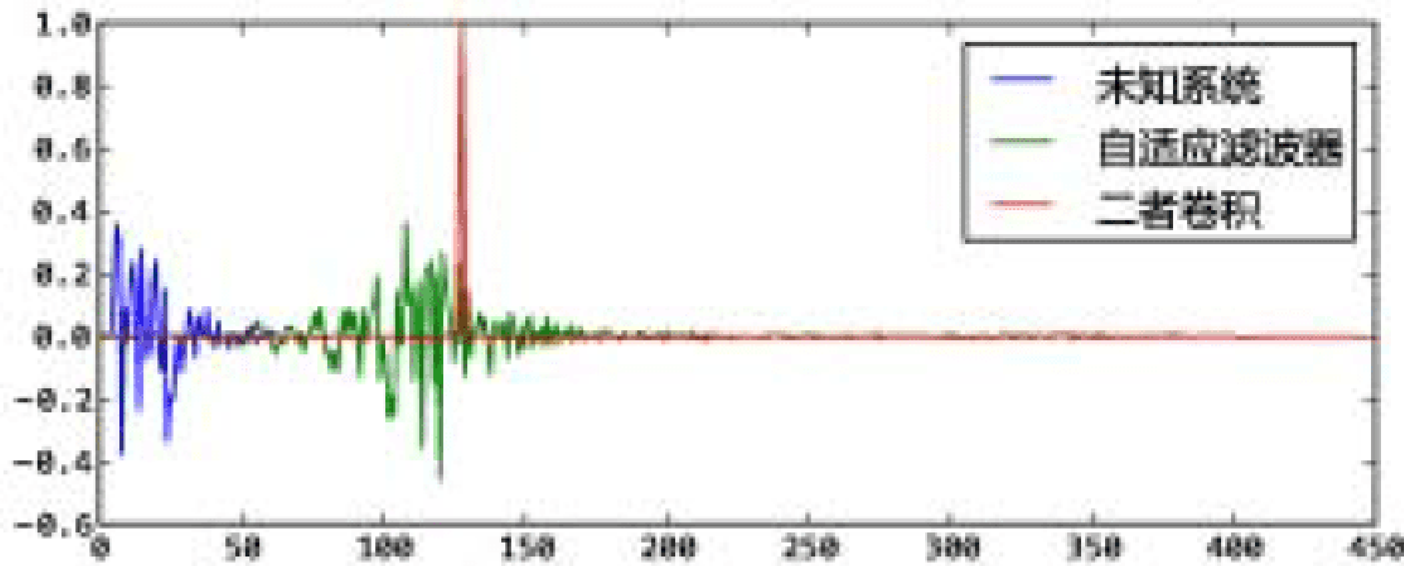


图 17-8　未知系统和自适应滤波器的级联(卷积)近似为标准延迟

图 17-9 显示了未知系统的频率响应(蓝色)和自适应滤波器的频率响应(绿色)，我们看到二者正好相反，也就是说自适应滤波器均衡了未知系统对信号的影响(见文前彩插)。

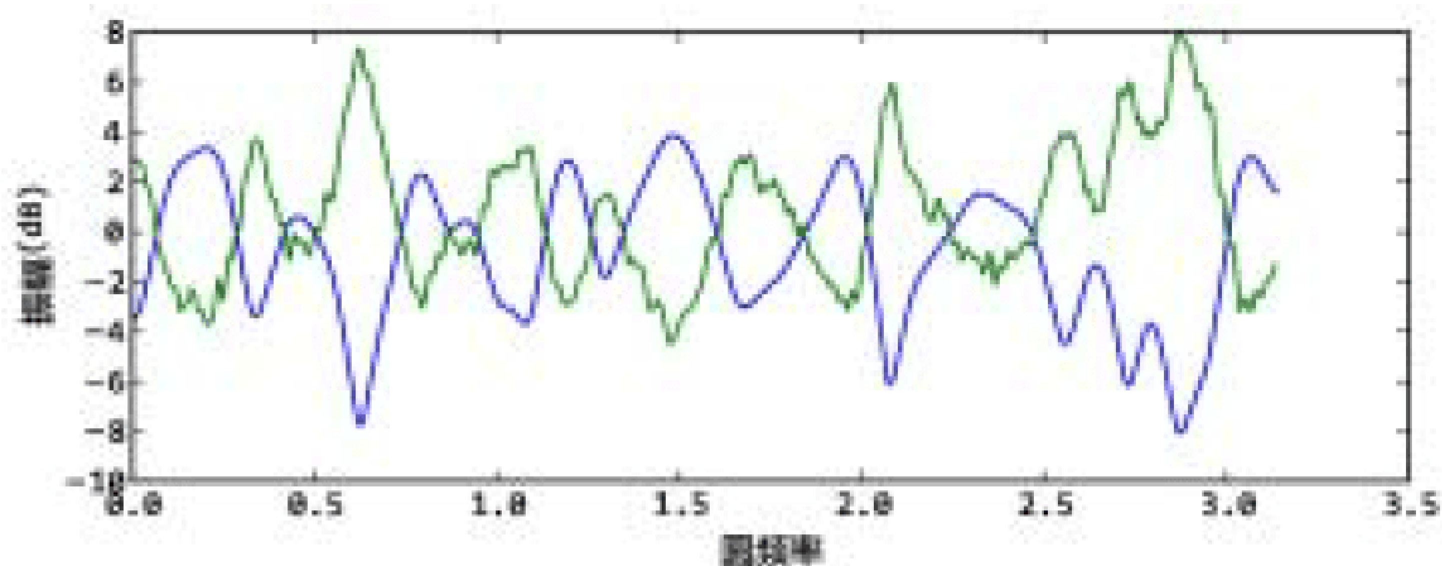


图 17-9　未知系统和自适应滤波器的频率响应正好相反

17.3.3　卷积逆运算

虽然卷积运算最终能归结为简单的加法和乘法运算，然而卷积的逆运算就不是很容易计算了。我们知道，两个线性系统 h1 和 h2 的级联 h3 可以用它们的脉冲响应的卷积计算求得，而所谓卷积的逆运算，可以想象为已知 h3 和 h1，求一个 h2 使它和 h1 级联之后正好等于 h3。

根据卷积的计算公式可知，如果 h1 的长度为 100，h3 的长度为 199，那么 h2 的长度也为 100，因为 h2 的每个系数都是未知的，于是就有 100 个未知数，而这 100 个未知数需要满足 199 个线性方程：h3 中的每个系数都有一个方程与之对应。由于方程数大于未知数的个数，显然对于任意的 h1 和 h3，并不能保证有一个 h2 使得它和 h1 的卷积正好等于 h3。

既然不能精确求解，那么卷积的逆运算就变成了一个误差最小化的优化问题。用自适应滤波器计算卷积的逆运算和计算信号均衡类似，将白噪声 x 输入到 h1 中得到信号 u，将 x 输入到 h3 中得到信号 d，然后使用 u 作为参照信号，d 作为目标信号进行 NLMS 计算，最终收敛后的自适应滤波器的系数就是 h2。

下面的程序模拟了这一过程，图 17-10 是程序的计算结果(见文前彩插)。



Inverse_convolve.py
自适应滤波器求卷积逆运算

```
from nlms_numpy import nlms
from scipy import signal
from nlms_common import *

def inverse_convolve(h1, h3, length):
    x = np.random.standard_normal(10000)
    u = signal.lfilter(h1, 1, x)
    d = signal.lfilter(h3, 1, x)
```

```
    h = np.zeros(length, np.float64)
    nlms(u, d, h, 0.1)
    return h

h1 = np.fromfile("h1.txt", sep="\n")
h1 /= np.max(h1)
h3 = np.fromfile("h3.txt", sep="\n")
h3 /= np.max(h3)

pl.rc('legend', fontsize=10)
pl.subplot(411)
pl.plot(h3, label="h3")
pl.plot(h1, label="h1")
pl.legend()
pl.gca().set_yticklabels([])
for idx, length in enumerate([128, 256, 512]):
    pl.subplot(412+idx)
    h2 = inverse_convolve(h1, h3, length)
    pl.plot(np.convolve(h1, h2)[:len(h3)], label="h1*h2(%s)" % length)
    pl.plot(h3, label="h3")
    pl.legend()
    pl.gca().set_yticklabels([])
    pl.gca().set_xticklabels([])

pl.show()
```

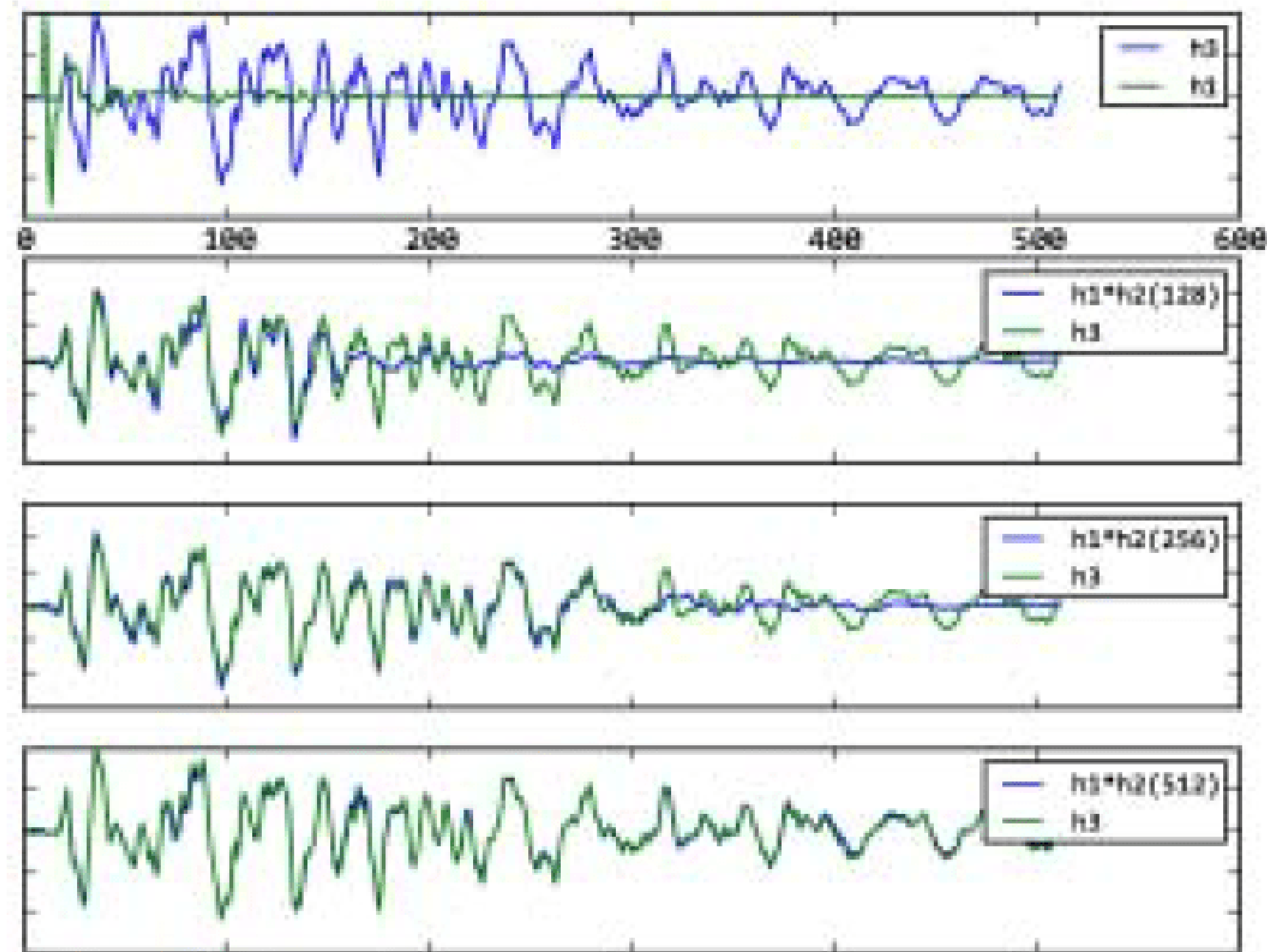


图 17-10　卷积逆运算结果演示

程序中的 h1 和 h3 从文本文件中读取而得，它们是 ANC(能动噪声控制)系统中实际测量的

脉冲响应。如果能找到一个 h2 满足卷积条件，就能够有效地进行噪声控制。

程序计算出 h2 的长度分别为 128、256、512 时的结果，可以看出 h2 的长度越长，结果越精确。

17.4 用 C 语言加速 NLMS 运算

在用 NumPy 实现 NLMS 算法的程序中，需要在 Python 级别对输入信号的每个取样值进行循环，显然这样做的运算效率并不高。为了让所有的循环都在 C 语言中完成，我们需要用 C 语言重写整个 nlms 函数。幸好 NLMS 算法并不复杂，用 C 语言编写也不会增加多少代码量。本节使用上一章介绍的方法，在 C 语言级别实现 NLMS 算法，并且通过 Python 调用它们。

17.4.1 用 SWIG 编写扩展模块

用 SWIG 可以很方便地为 C 语言函数制作封装函数，让它们能在 Python 环境中调用。下面让我们看看如何使用 SWIG 制作计算 NLMS 的扩展模块。



cnlms/cnlms.cpp
用 C 语言实现的 NLMS 算法

```
void cnlms(double x[], int nx, double d[], int nd,
    double h[], int nh, double step, double u[], int nu)
{
 int i, j, count;
 double s, e, power=0;
 double *px;

    // 取 nx、nd、nu 的最小值
    count = nx<nd?nx:nd;
    count = count<nu?count:nu;

    // 开始的 nh 个取样不更新滤波器系数
    for(i=0;i<nh;i++)
    {
          power += x[i] * x[i];
            u[i] = 0;
    }

    for(i=nh;i<count;i++)
    {
          s = 0;
          px = &x[i];
```

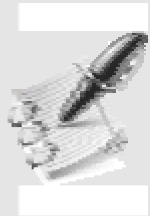
```
            for(j=0;j<nh;j++)
            {
                s += (*px--) * h[j];
            }
            u[i] = s;
            e = d[i] - s;

            px = &x[i];
            for(j=0;j<nh;j++)
            {
                h[j] += step * e * (*px--) / power;
            }

            power -= x[i-nh+1] * x[i-nh+1];
              if(i<count-1)
                power += x[i+1] * x[i+1];
        }
    }
```

这里不对 cnlms()的内容进行详细分析，只看看它的调用形式。其中，x 是长度为 nx 的参照信号；d 是长度为 nd 的目标信号；h 是长度为 nh 的未知系统的系数；step 是更新系数；u 用来保存滤波器的输出信号，其长度为 nu。

在 cnlms()中，我们看不到任何与 NumPy 数组有关的定义。将 NumPy 数组转换成 cnlms()可以处理的 C 语言数组的工作由接口定义文件“cnlms.i”完成：



cnlms/cnlms.i
“cnlms.cpp”的 SWIG 接口文件

```
%module nlms_swig
%{
#define SWIG_FILE_WITH_INIT
#include "cnlms.h"
%}
%include "numpy.i"
%init %{
    import_array();
%}

void cnlms(
    double * IN_ARRAY1, int DIM1,  // x
    double * IN_ARRAY1, int DIM1,  // d
    double * INPLACE_ARRAY1, int DIM1, // h
    double step,
```

```
    double * ARGOUT_ARRAY1, int DIM1 // u
);

%pythoncode %{
def nlms(x, d, h, step):
    n = min(len(x), len(d))
    return _nlms_swig.cnlms(x, d, h, step, n)
%}
```

在接口文件中包含了定义各种 NumPy 类型映射的“numpy.i”文件。我们使用了三种不同的类型映射对 cnlms()的参数进行了转换：

- 由于参照信号 x 和目标信号 d 在整个计算过程中不会发生变化，因此使用(IN_ARRAY1, DIM1)将它们定义为输入用的数组参数。
- 滤波器的系数 h 会被更新，因此使用(INPLACE_ARRAY1, DIM1)将它转换为更新用的数组参数。
- cnlms()返回滤波器的输出数组 u，因此用(ARGOUT_ARRAY1, DIM1)将它定义为返回用的数组。

这样一来，SWIG 生成的封装函数所接收的参数如下：

```
cnlms(x, d, h, step, nu)
```

并且它返回一个长度为 nu 的数组以表示滤波器的输出信号。显然需要用户指定 nu 是多余的，因为 nu 可以由 x 和 d 的长度决定。因此在接口文件中我们用%pythoncode 命令定义了一个新的函数 nlms()，这段程序会被 SWIG 输出到自动生成的“cnlms.py”中。

最后我们制作了一个“setup.py”文件，运行“setup.py build_ext --inplace”即可编译出用 C 语言编写的 NLMS 扩展模块。

为了比较 NumPy 版本和 SWIG 版本的 NLMS 程序的输出是否一致，我们制作了一个简单的脚本来进行测试：



test_nlms.py
验证 NumPy 版本和 SWIG 版本的 NLMS 程序的输出

```
import numpy as np
from nlms_numpy import nlms as numpy_nlms
from nlms_swig import nlms as swig_nlms
from nlms_weave import nlms as weave_nlms
x = np.random.random(100)
d = np.random.random(100)
h = np.zeros(10)
u1 = numpy_nlms(x, d, h, 0.1)
```

```
h = np.zeros(10)
u2 = swig_nlms(x, d, h, 0.1)
h = np.zeros(10)
u3 = weave_nlms(x, d, h, 0.1)

print np.sum((u1-u2)**2)
print np.sum((u1-u3)**2)
```

测试程序的输出约为 3e-31，因此可以认为两个程序的运行结果是一致的。请读者将前面介绍的几个模拟程序的载入语句修改为“from nlms_swig import nlms”，并测试程序的运行时间，看看用 C 语言编写的 NLMS 算法能带来多少倍的速度提升。

17.4.2 用 Weave 嵌入 C++程序

用 SWIG 编写扩展模块比较麻烦，需要准备接口文件、编写“setup.py”脚本。对于 NLMS 这样简单的程序显得有些大材小用。更简单的办法是使用 Weave 直接将 C++语言的程序嵌入到 Python 程序中。下面是完整的源程序：



nlms_weave.py
用 Weave 将 NLMS 的 C++语言计算程序嵌入到 Python 中

```
import numpy as np
import scipy.weave as weave

def nlms(x, d, h, step):
    code = """
    int i, j, count;
    int nh = Nh[0];
    double s, e, power=0;
    double *px;
    count = Nu[0];
    for(i=0;i<nh;i++)
    {
        power += x(i) * x(i);
        u(i) = 0;
    }
    for(i=nh;i<count;i++)
    {
        s = 0;
        px = &x(i);
        for(j=0;j<nh;j++)
        {
            s += (*px--) * h(j);
```

```
        }
        u(i) = s;
        e = d(i) - s;

        px = &x(i);
        for(j=0;j<nh;j++)
        {
            h(j) += step * e * (*px--) / power;
        }
        power -= x(i-nh+1) * x(i-nh+1);
        if(i<count-1)
            power += x(i+1) * x(i+1);
    }
    """
    u = np.zeros(min(len(x), len(d))) ❶
    weave.inline(
        code,
        ['x','d','h','u','step'], ❷
        type_converters=weave.converters.blitz, ❸
        compiler="gcc"
    )
    return u
```

❶创建保存滤波器输出的数组，其长度取参照信号 x 和目标信号 d 中长度较短的那个。❷通过字符串将当前名称空间中的 Python 变量转换成 C++语言中的变量。❸我们使用 blitz 类处理 NumPy 数组，因此使用 weave.converters.blitz 作为类型转换器。

在嵌入的 C++程序中，使用 Nx[0]、Nd[0]、Nh[0]、Nu[0]获得各个数组的第 0 轴的长度。使用圆括号访问数组的下标，例如 x(i)、h(j)。

第 18 章

单摆和双摆模拟

本章首先介绍单摆和双摆系统的公式推导，然后通过 odeint()对其进行数值求解并制作动画演示程序。

18.1 单摆模拟

如图 18-1 所示，有一根不可伸长、质量不计的细棒，上端固定，下端系一质点，这样的装置叫做单摆。

根据牛顿力学定律，我们可以列出如下微分方程：

$$\frac{d^2\theta}{dt^2}+\frac{g}{l}\sin\theta=0$$

其中，θ 为单摆的摆角，l 为单摆的长度，g 为重力加速度。

此微分方程的符号解无法直接求出，因此只能调用 odeint()对其求数值解。

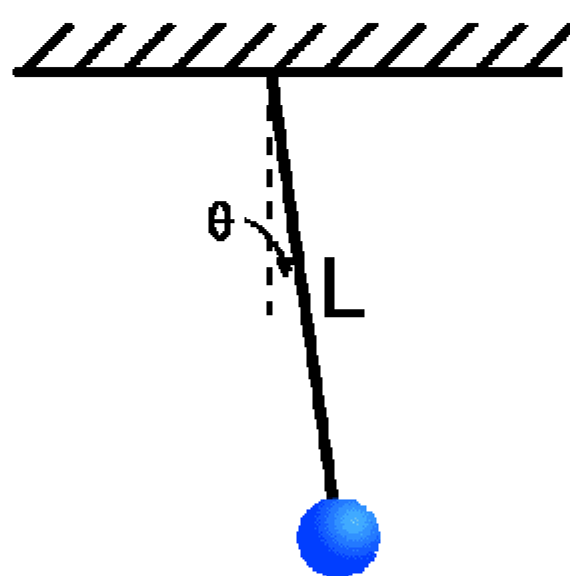


图 18-1　单摆装置示意图

odeint()的调用参数如下：

```
odeint(func, y0, t, ...)
```

其中，func 是 Python 的一个函数对象，用来计算微分方程组中每个未知函数的导数；y0 为微分方程组中每个未知函数的初始值；t 为需要进行数值求解的时间点。它返回的是一个二维数组 result，其第 0 轴的长度为 t 的长度，第 1 轴的长度为变量的个数，因此 result[:,i]为第 i 个未知函数的解。

计算微分的 func 函数的调用参数为 func(y, t)，其中 y 是一个数组，为每个未知函数在 t 时刻的值，而 func 的返回值是每个未知函数在 t 时刻的导数。

odeint()要求每个微分方程只包含一阶导数，因此我们需要对前面的微分方程进行如下变形：

$$\frac{d\theta(t)}{dt}=v(t)$$

$$\frac{dv(t)}{dt}=-\frac{g}{l}\sin\theta(t)$$

下面是计算单摆轨迹的程序，摆角和时间的关系如图 18-2 所示。

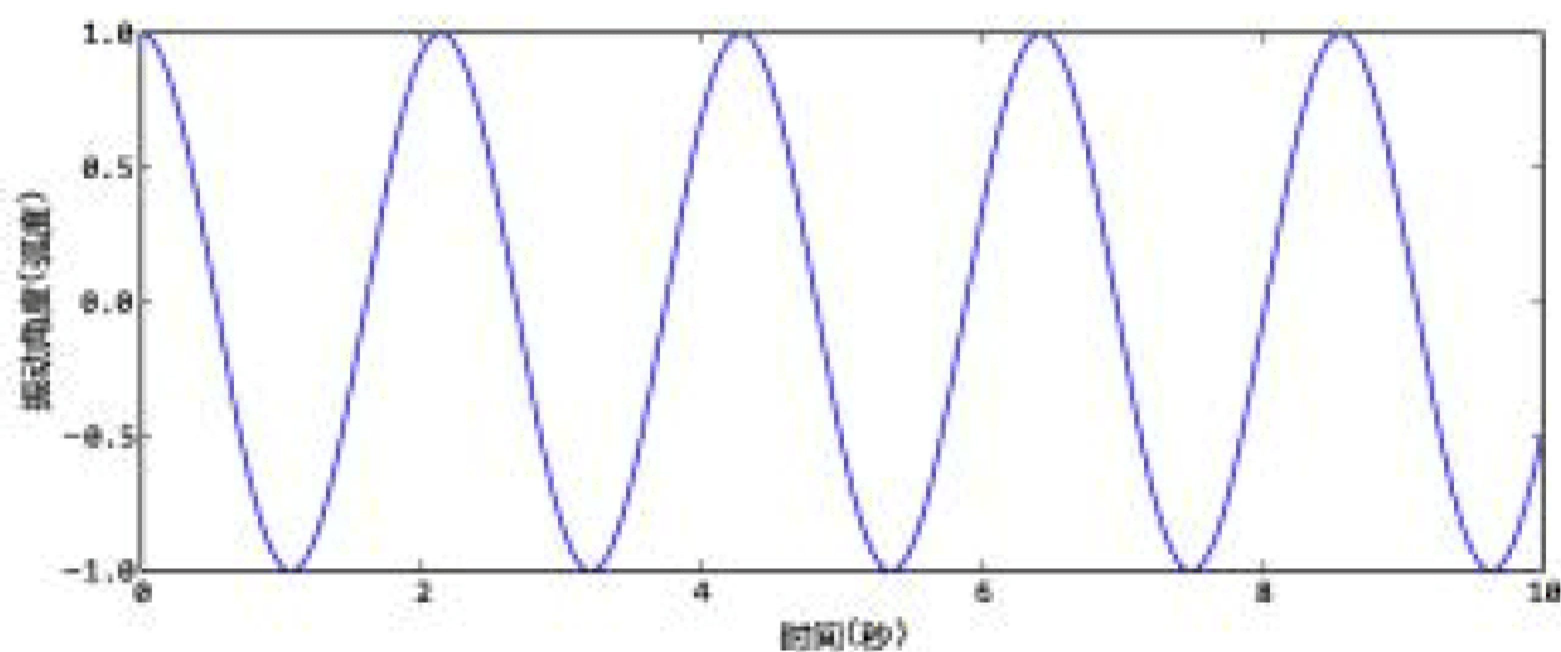


图 18-2　初始角度为 1 弧度的单摆摆动角度和时间的关系



simple_pendulum_odeint.py
用 odeint 计算单摆轨迹

```
from math import sin
import numpy as np
from scipy.integrate import odeint

g = 9.8

def pendulum_equations(w, t, l):
    th, v = w
    dth = v
    dv  = - g/l * sin(th)
    return dth, dv

if __name__ == "__main__":
    import pylab as pl
    t = np.arange(0, 10, 0.01)
    track = odeint(pendulum_equations, (1.0, 0), t, args=(1.0,))
    pl.plot(t, track[:, 0])
    pl.xlabel(u"时间(秒)")
    pl.ylabel(u"震度角度(弧度)")
    pl.show()
```

odeint()还有一个 args 参数，参数值为一个元组，这些值都会作为额外的参数传递给计算导数的函数。程序使用这种方式将单摆的长度传递给 pendulum_equations()。

18.1.1　小角度时的摆动周期

高中物理课上讲过，当最大摆动角度很小时，单摆的摆动周期可以使用如下公式计算：

$$T_0 = 2\pi\sqrt{\frac{l}{g}}$$

这是因为当 $\theta << 1$ 时，$\sin\theta \approx \theta$，这样微分方程就变成了：

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{g}{l}\theta = 0$$

此微分方程的解是一个简谐振动方程，很容易计算其摆动周期。下面我们用 SymPy 对这个微分方程进行符号求解：

```
>>> from sympy import symbols, Function, dsolve
>>> t,g,l = symbols("t,g,l",real=True) # 分别表示时间、重力加速度和长度
>>> y = Function("y") # 摆角函数用 y(t)表示
>>> dsolve(y(t).diff(t,2) + g/l*y(t), y(t))
```

$$y(t) = C_1 \sin(\frac{t\sqrt{g}}{\sqrt{l}}) + C_2 \cos(\frac{t\sqrt{g}}{\sqrt{l}})$$

可以看到，简谐振动方程的解是由两个频率相同的三角函数构成的，其周期为 $2\pi\sqrt{\frac{l}{g}}$。

18.1.2 大角度时的摆动周期

但是当初始摆角增大时，上述近似处理会带来无法忽视的误差。下面让我们看看如何用数值计算的方法求出单摆在任意初始摆角时的摆动周期。



simple_pendulum_period.py
数值法求单摆摆动周期

要计算摆动周期，只需要计算从最大摆角到 0 摆角所需的时间，摆动周期是此时间的 4 倍。为了计算出这个时间值，首先需要定义一个函数 pendulum_th()，计算任意时刻的摆角：

```
def pendulum_th(t, l, th0):
    track = odeint(pendulum_equations, (th0, 0), [0, t], args=(l,))
    return track[-1, 0]
```

pendulum_th()计算长度为l、初始角度为th0的单摆在时刻t的摆角。此函数仍然使用odeint()进行微分方程组求解，只是我们只需要计算时刻 t 的摆角，因此传递给 odeint()的时间序列为 [0, t]。odeint()内部会对时间进行细分，以保证最终的解是正确的。

接下来只需找到第一个使 pendulum_th()的值为 0 的时间即可。这相当于对 pendulum_th()求解，可以使用 scipy.optimize.fsolve()对这种非线性方程进行求解：

```
def pendulum_period(l, th0):
    t0 = 2*np.pi*sqrt( l/g ) / 4
    t = fsolve( pendulum_th, t0, args = (l, th0) )
    return t*4
```

和 odeint()一样，我们通过 fsolve()的 args 参数将额外的参数传递给 pendulum_th()。fsolve()求解时需要一个初始值尽量接近真实值的解，用小角度单摆的周期的 1/4 作为这个初始值是一个很不错的选择。下面利用 pendulum_period()计算出初始摆动角度从 0° 到 90° 的摆动周期：

```
ths = np.arange(0, np.pi/2.0, 0.01)
periods = [pendulum_period(1, th) for th in ths]
```

为了验证结果的正确性，可以从维基百科中找到摆动周期的精确解：

$$T = 4\sqrt{\frac{l}{g}}K(\sin\frac{\theta_0}{2})$$

其中的函数 K 为第一类完全椭圆积分函数，其定义如下：

$$K(k) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}}$$

可以用 scipy.special.ellipk()计算此函数的值：

```
periods2 = 4*sqrt(1.0/g)*ellipk(np.sin(ths/2)**2) # 计算单摆周期的精确值
```

图 18-3 比较了这两种计算方法，我们看到它们的结果是完全一致的(见文前彩插)。

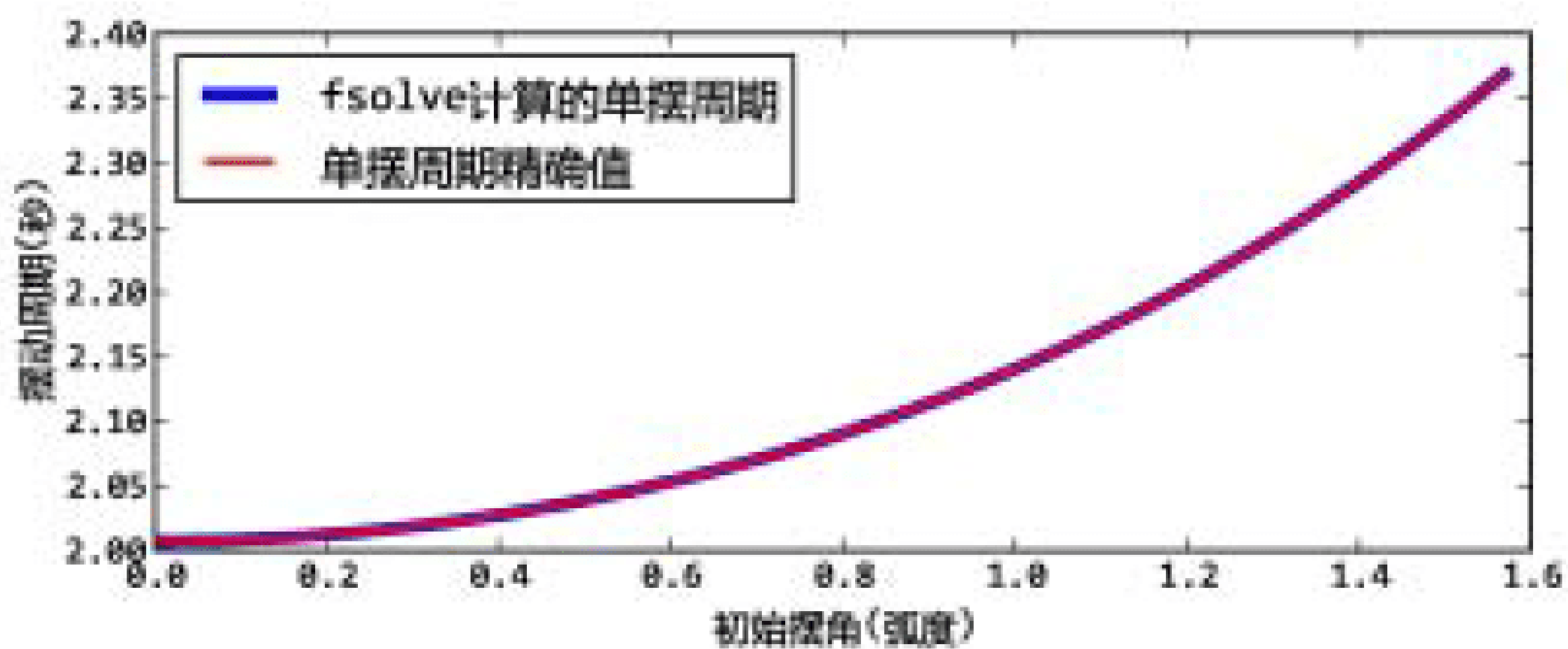


图 18-3　单摆的摆动周期和初始角度的关系

18.2　双 摆 模 拟

接下来让我们看看如何对双摆系统进行模拟。双摆系统的示意图如图 18-4 所示。两根长度为 L_1 和 L_2 的无质量细棒的顶端有质量分别为 m_1 和 m_2 的两个球，初始角度为 θ_1 和 θ_2，要求计算从此初始状态释放之后两个球的运动轨迹。

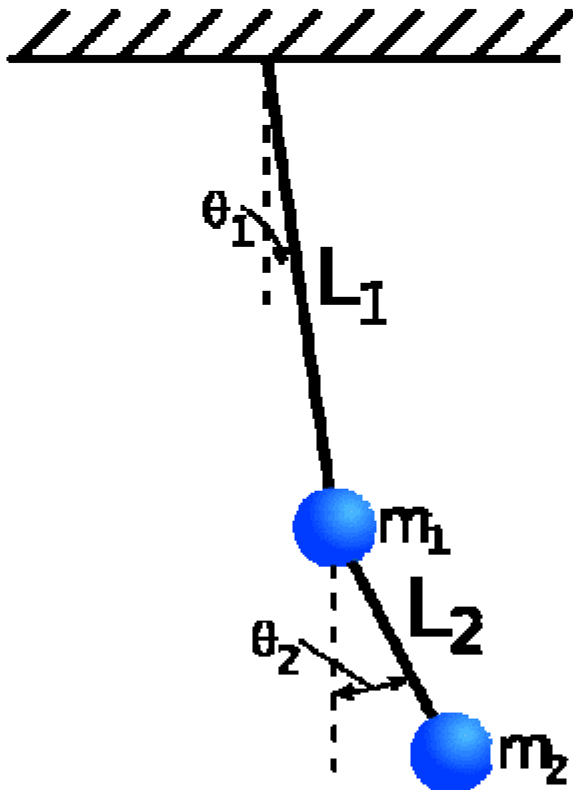


图 18-4　双摆装置示意图

18.2.1　公式推导

本节首先介绍如何利用拉格朗日力学获得双摆系统的微分方程组。

拉格朗日力学

拉格朗日力学是分析力学中的一种。它于 1788 年由拉格朗日创立，拉格朗日力学是对经典力学的一种新的理论表述。

经典力学最初的表述形式由牛顿建立，它着重于分析位移、速度、加速度、力等矢量间的关系，又称为矢量力学。拉格朗日引入了广义坐标的概念，又运用达朗贝尔原理，求得与牛顿第二定律等价的拉格朗日方程。不仅如此，拉格朗日方程还具有更普遍的意义，适用范围更广泛。此外，选取恰当的广义坐标，可以大大地简化拉格朗日方程的求解过程。

假设细棒 L_1 连接的球体的坐标为 x_1 和 y_1，L_2 连接的球体的坐标为 x_2 和 y_2，那么 x_1、y_1、x_2、y_2 和两个角度 θ_1、θ_2 之间有如下关系：

$$x_1 = L_1 \sin(\theta_1), y_2 = -L_1 \cos(\theta_1)$$

$$x_2 = L_1 \sin(\theta_1) + L_2 \sin(\theta_2), y_2 = -L_1 \cos(\theta_1) - L_2 \cos(\theta_2)$$

根据拉格朗日力学公式：

$$\mathcal{L} = T - V$$

其中，T 为系统的动能，V 为系统的势能，可以得到如下公式：

$$\mathcal{L} = \frac{m_1}{2}(\dot{x}_1^{\ 2} + \dot{y}_1^{\ 2}) + \frac{m_2}{2}(\dot{x}_2^{\ 2} + \dot{y}_2^{\ 2}) - m_1 g y_1 - m_2 g y_2$$

其中正号的项为两个小球的动能，负号的项为两个小球的势能。
将前面的坐标和角度之间的关系公式代入并整理可得：

$$\mathcal{L} = \frac{m_1 + m_2}{2} L_1^{\ 2} \dot{\theta}_1^{\ 2} + \frac{m_2}{2} L_2^{\ 2} \dot{\theta}_2^{\ 2} + m_2 L_1 L_2 \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2) + (m_1 + m_2) g L_1 \cos(\theta_1) + m_2 g L_2 \cos(\theta_2)$$

对于变量 θ_1 的拉格朗日方程：

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2} - \frac{\partial \mathcal{L}}{\partial \theta_1} = 0$$

得到：

$$L_1[(m_1 + m_2) L_1 \ddot{\theta}_1 + m_2 L_2 \cos(\theta_1 - \theta_2)\ddot{\theta}_2 + m_2 L_2 \sin(\theta_1 - \theta_2)\dot{\theta}_2^2 + (m_1 + m_2) g \sin(\theta_1)] = 0$$

对于变量 θ_2 的拉格朗日方程：

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2} - \frac{\partial \mathcal{L}}{\partial \theta_2} = 0$$

得到：

$$m_2 L_2\ [L_2 \ddot{\theta}_2 + L_1 \cos(\theta_1 - \theta_2)\ddot{\theta}_1 - L_1 \sin(\theta_1 - \theta_2)\dot{\theta}_1^{\ 2} + g \sin(\theta_2)] = 0$$

这一计算过程可以用 SymPy 进行推导：



double_pendulum_solver.py
用 SymPy 推导双摆的常微分方程

```
from sympy import *
from sympy import Derivative as D

var("x1 x2 y1 y2 l1 l2 m1 m2 th1 th2 dth1 dth2 ddth1 ddth2 t g tmp")

sublist = [
(D(th1(t), t, t), ddth1),
(D(th1(t), t), dth1),
(D(th2(t), t, t), ddth2),
(D(th2(t),t), dth2),
```

```
    (th1(t), th1),
    (th2(t), th2)
    ]

    x1 = l1*sin(th1(t))
    y1 = -l1*cos(th1(t))
    x2 = l1*sin(th1(t)) + l2*sin(th2(t))
    y2 = -l1*cos(th1(t)) - l2*cos(th2(t))

    vx1 = diff(x1, t)
    vx2 = diff(x2, t)
    vy1 = diff(y1, t)
    vy2 = diff(y2, t)

    # 拉格朗日力学公式
    L = m1/2*(vx1**2 + vy1**2) + m2/2*(vx2**2 + vy2**2) - m1*g*y1 - m2*g*y2

    # 拉格朗日方程
    def lagrange_equation(L, v):
        dvt = D(v(t), t)
        a = L.subs(dvt, tmp).diff(tmp).subs(tmp, dvt) ❶
        b = L.subs(dvt, tmp).subs(v(t), v).diff(v).subs(v, v(t)).subs(tmp, dvt) ❷
        c = a.diff(t) - b
        c = c.subs(sublist)
        c = trigsimp(simplify(c))
        c = collect(c, [th1,th2,dth1,dth2,ddth1,ddth2])
        return c

    eq1 = lagrange_equation(L, th1)
    eq2 = lagrange_equation(L, th2)
    print eq1
    print eq2
```

执行此程序之后，eq1 对应于 θ_1 的拉格朗日方程，eq2 对应于 θ_2 的拉格朗日方程。

由于 SymPy 只能对符号变量求导数，即只能计算 D(L, t)，而不能计算 D(f, v(t))。❶因此在求偏导数之前，将偏导数变量置换为一个 tmp 变量，然后对 tmp 变量求导数，例如下面的程序对 D(v(t), t)求偏导数，即计算 $\partial L/\partial \dot{v}$：

```
    L.subs(D(v(t), t), tmp).diff(tmp).subs(tmp, D(v(t), t))
```

❷而在计算 $\partial L/\partial v$ 时，需要将 v(t)替换为 v 之后再进行微分计算。由于将 v(t)替换为 v 的同时，会将 D(v(t), t)中的也进行替换，这不是我们希望的结果，因此先将 D(v(t), t)替换为 tmp，微分计算完毕之后再替换回去：

```
    L.subs(D(v(t), t), tmp).subs(v(t), v).diff(v).subs(v, v(t)).subs(tmp, D(v(t), t))
```

最后得到 eq1 和 eq2 的值为：

```
>>> eq1
ddth1*(m1*l1**2 + m2*l1**2) +
ddth2*(l1*l2*m2*cos(th1)*cos(th2) + l1*l2*m2*sin(th1)*sin(th2)) +
dth2**2*(l1*l2*m2*cos(th2)*sin(th1) - l1*l2*m2*cos(th1)*sin(th2)) +
g*l1*m1*sin(th1) + g*l1*m2*sin(th1)
>>> eq2
ddth1*(l1*l2*m2*cos(th1)*cos(th2) + l1*l2*m2*sin(th1)*sin(th2)) +
dth1**2*(l1*l2*m2*cos(th1)*sin(th2) - l1*l2*m2*cos(th2)*sin(th1)) +
g*l2*m2*sin(th2) + ddth2*m2*l2**2
```

结果看上去挺复杂，其实只要运用如下的三角公式，就和前面的结果一致了：

$$\sin(x+y)=\sin x\cos y+\cos x\sin y$$
$$\cos(x+y)=\cos x\cos y-\sin x\sin y$$
$$\sin(x-y)=\sin x\cos y-\cos x\sin y$$
$$\cos(x-y)=\cos x\cos y+\sin x\sin y$$

18.2.2 微分方程的数值解

接下来要做的事情就是对如下的微分方程组求数值解：

$$(m_1+m_2)L_1\ \ddot{\theta}_1+m_2L_2\cos(\theta_1-\theta_2)\ddot{\theta}_2+m_2L_2\sin(\theta_1-\theta_2)\dot{\theta}_2{}^2+(m_1+m_2)g\sin(\theta_1)=0$$

$$L_2\ \ddot{\theta}_2+L_1\cos(\theta_1-\theta_2)\ddot{\theta}_1-L_1\sin(\theta_1-\theta_2)\dot{\theta}_1{}^2+g\sin(\theta_2)=0$$

由于方程中包含二阶导数，因此无法直接使用 odeint()进行数值求解，我们很容易将其改写为 4 个一阶微分方程组，4 个未知变量为θ_1、θ_2、v_1、v_2，其中，v_1、v_2为两个细棒转动的角速度。

$$\dot{\theta}_1=v_1$$
$$\dot{\theta}_2=v_2$$
$$(m_1+m_2)L_1\ \dot{v}_1+m_2L_2\cos(\theta_1-\theta_2)\dot{v}_2+m_2L_2\sin(\theta_1-\theta_2)\dot{\theta}_2{}^2+(m_1+m_2)g\sin(\theta_1)=0$$
$$L_2\ \dot{v}_2+L_1\cos(\theta_1-\theta_2)\dot{v}_1-L_1\sin(\theta_1-\theta_2)\dot{\theta}_1{}^2+g\sin(\theta_2)=0$$

下面的程序对此微分方程组进行数值求解。



double_pendulum_odeint.py
用 odeint 求解双摆系统

```
g = 9.8
class DoublePendulum(object):
    def __init__(self, m1, m2, l1, l2):
        self.m1, self.m2, self.l1, self.l2 = m1, m2, l1, l2
        self.init_status = np.array([0.0,0.0,0.0,0.0])

    def equations(self, w, t): ❶
        """
        微分方程公式
        """
        m1, m2, l1, l2 = self.m1, self.m2, self.l1, self.l2
        th1, th2, v1, v2 = w
        dth1 = v1
        dth2 = v2

        #eq of th1
        a = l1*(m1+m2)  # dv1 parameter
        b = m2*l2*cos(th1-th2) # dv2 paramter
        c = m2*l2*sin(th1-th2)*dth2*dth2 + (m1+m2)*g*sin(th1)

        #eq of th2
        d = l1*cos(th1-th2) # dv1 parameter
        e = l2 # dv2 parameter
        f = -l1*sin(th1-th2)*dth1*dth1 + g*sin(th2)

        dv1, dv2 = np.linalg.solve([[a,b],[d,e]], [-c,-f])  ❷

        return np.array([dth1, dth2, dv1, dv2])
```

❶DoublePendulum 类的 equations()用于计算各个未知函数的导数，输入参数 w 数组中的变量依次为 th1、th2、v1 和 v2，它们分别表示上球角度、下球角度、上球角速度和下球角速度。

返回值为每个变量的导数 dth1、dth2、dv1 和 dv2，它们分别表示上球角速度、下球角速度、上球角加速度和下球角加速度。其中，dth1 和 dth2 很容易计算，它们直接等于传入的角速度变量。

为了计算 dv1 和 dv2，需要将微分方程组变形为如下格式：

$$\dot{v}_1 = \ldots, \dot{v}_2 = \ldots$$

由于两个微分方程对于 $\dot{v}_1$ 和 $\dot{v}_2$ 来说是两个如下形式的一次方程：

$$a\dot{v}_1 + b\dot{v}_2 + c = 0, d\dot{v}_1 + e\dot{v}_2 + f = 0$$

因此可以求出其中的系数 a、b、c、d、e、f，❷然后调用 linalg.solve()解出 $\dot{v}_1$ 和 $\dot{v}_2$。

```
def double_pendulum_odeint(pendulum, ts, te, tstep):
    """
```

```
    对双摆系统的微分方程组进行数值求解，返回两个小球的 X-Y 坐标
    """
    t = np.arange(ts, te, tstep)
    track = odeint(pendulum.equations, pendulum.init_status, t) ❸
    th1_array, th2_array = track[:,0], track[:, 1]
    l1, l2 = pendulum.l1, pendulum.l2
    x1 = l1*np.sin(th1_array)
    y1 = -l1*np.cos(th1_array)
    x2 = x1 + l2*np.sin(th2_array)
    y2 = y1 - l2*np.cos(th2_array)
    #将最后的状态赋给 pendulum
    pendulum.init_status = track[-1,:].copy() ❹
    return x1, y1, x2, y2 ❺
```

在 double_pendulum_odeint()中，❸调用 odeint()对双摆的方程组求数值解。❹将 odeint()得到的最终的小球状态保存到 pendulum.init_status 中，作为下一次调用 odeint()的初始值，因此多次调用 double_pendulum_odeint()可以生成连续的运动轨迹。❺函数返回 4 个数组，分别是两个小球的 X-Y 轴的坐标。

最后是主程序部分，我们使用小角度和大角度的初始值分别计算双摆的摆动轨迹。小初始角度时小球的运动轨迹如图 18-5 所示(见文前彩插)。大初始角度时小球的运动轨迹如图 18-6 所示(见封三彩插)。可以看出，当初始角度很大时，摆动出现混沌现象。

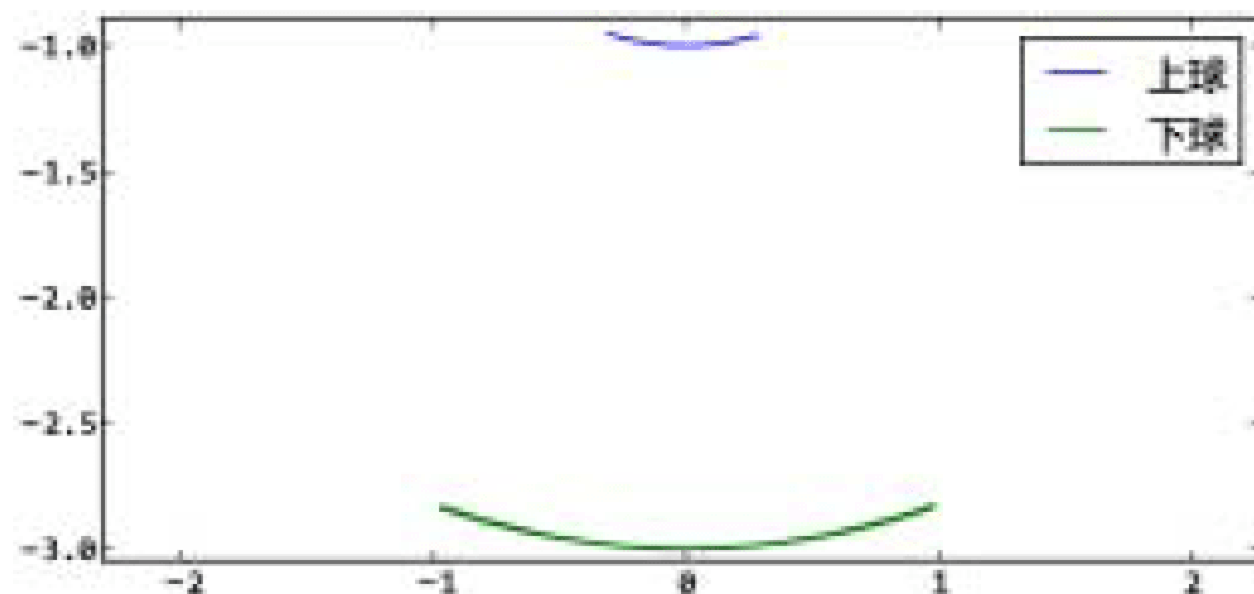


图 18-5　小初始角度时双摆的摆动轨迹

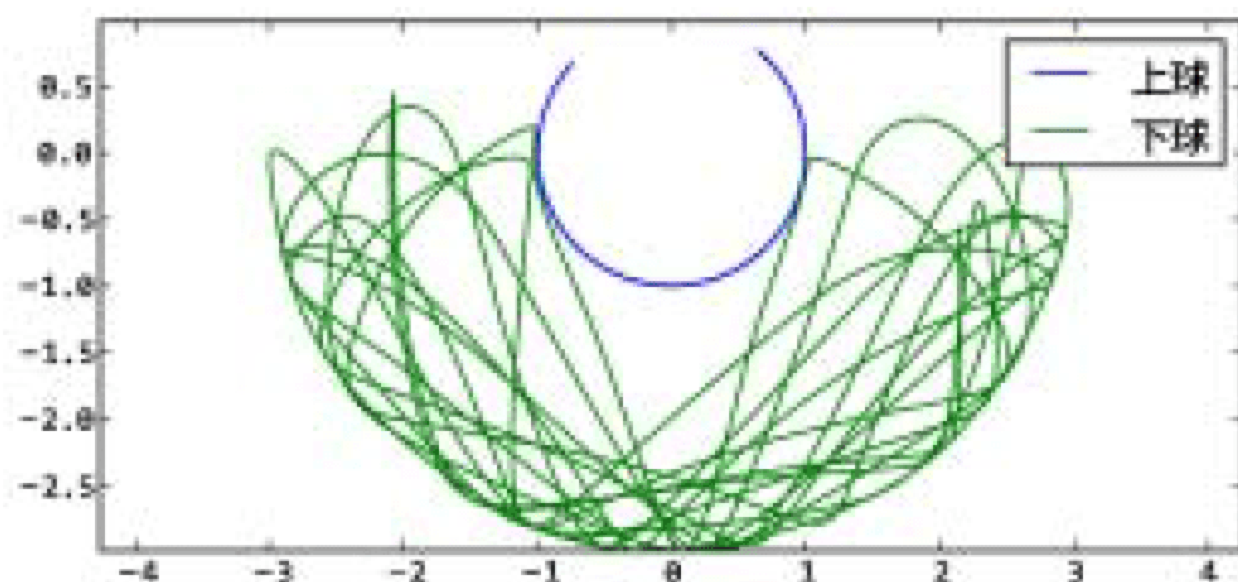


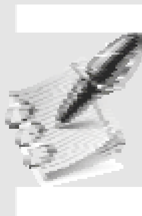
图 18-6　大初始角度时双摆的摆动轨迹呈现混沌现象

18.2.3　动画演示

有了计算双摆运动轨迹的程序之后，我们很容易使用前面介绍过的动画技术直观地演示双摆系统的运算结果。制作动画可以有多种选择：

- 使用 VPython 制作三维动画。
- 使用 Tkinter 或 wxPython 直接在界面上绘制动画。
- 使用 Enable、Chaco 或 matplotlib 制作动画。

本书提供使用 matplotlib 和 Enable 制作的双摆动画演示程序，它们调用上一节介绍的 double_pendulum_odeint()来计算双摆的运动轨迹。



double_pendulum_pylab_ani.py, double_pendulum_enable_ani.py
用 matplotlib 和 Enable 制作双摆的动画演示

使用 Enable 制作的动画演示程序的界面截图如图 18-7 所示。在此界面中，用户可以修改双摆的质量和长度，并且可以在动画框中按住并拖动鼠标，修改双摆的初始角度。

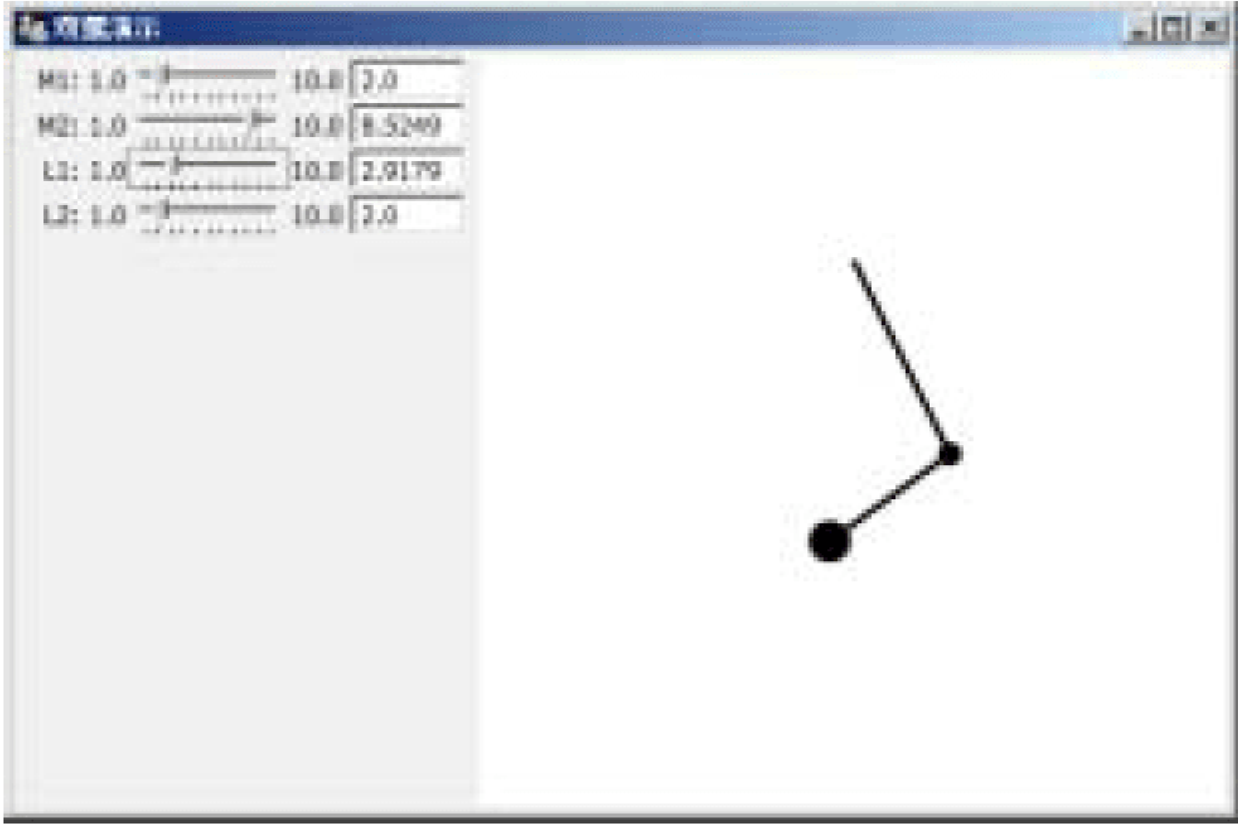


图 18-7　用 Enable 制作的双摆动画演示程序

第 19 章

分 形 几 何

自然界的很多事物，例如树木、云彩、山脉、闪电、雪花以及海岸线等，都呈现出传统几何学难以描述的形状。这些形状都有如下特性：

- 具有十分精细的不规则的结构。
- 整体与局部相似，例如一根树杈的形状和一棵树很像。

分形几何学就是用来研究这样一类几何形状的数学，借助计算机的高速计算和图像显示，我们可以更深入、更直观地研究分形几何。作为本书的最后一章，让我们看看如何使用 Python 绘制一些经典的分形图形。

19.1　Mandelbrot 集合

Mandelbrot(曼德布洛特)集合是在复平面上构成分形图案的点的集合，它可以用下面的复二次多项式定义：

$$f_c(z) = z^2 + c$$

其中，复数函数 $f_c(z)$的自变量为 z。c 是一个复数参数，对于每一个 c，从 z=0 开始对函数 $f_c(z)$进行迭代。序列$(0, f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), ...$的值要么延伸到无限大，要么只停留在有限半径的圆盘内。Mandelbrot 集合就是使以上序列不发散的所有参数 c 的集合。

从数学上来讲，Mandelbrot 集合是一个复数的集合。一个给定的复数 c 要么属于 Mandelbrot 集合，要么不是。但是用程序绘制 Mandelbrot 集合时不能进行无限次迭代，最简单的方法是使用逃逸时间(迭代次数)进行绘制，具体算法如下：

- 判断每次调用函数 $f_c(z)$后得到的结果是否在半径 R 之内，即复数的模是否小于 R。
- 记录下迭代结果的模值大于 R 时的迭代次数，也称为逃逸时间。
- 迭代最多进行 N 次。
- 不同迭代次数的点使用不同的颜色来绘制。

下面是绘制 Mandelbrot 集合的完整程序，绘制的图案如图 19-1 所示(见封三彩插)。



mandelbrot_python.py
绘制 Mandelbrot 集合

```
import numpy as np
import pylab as pl
import time
from matplotlib import cm

def iter_point(c): ❶
    z = c
    for i in xrange(1, 100): # 最多迭代 100 次
        if abs(z)>2: break # 半径大于 2 就认为逃逸
        z = z*z+c
    return i # 返回迭代次数

def draw_mandelbrot(cx, cy, d): ❷
    """
    绘制点(cx, cy)附近在正负 d 范围之内的 Mandelbrot
    """
    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:200j, x0:x1:200j]
    c = x + y*1j ❸

    start = time.clock()
    mandelbrot = np.frompyfunc(iter_point,1,1)(c).astype(np.float) ❹
    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1]) ❺
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

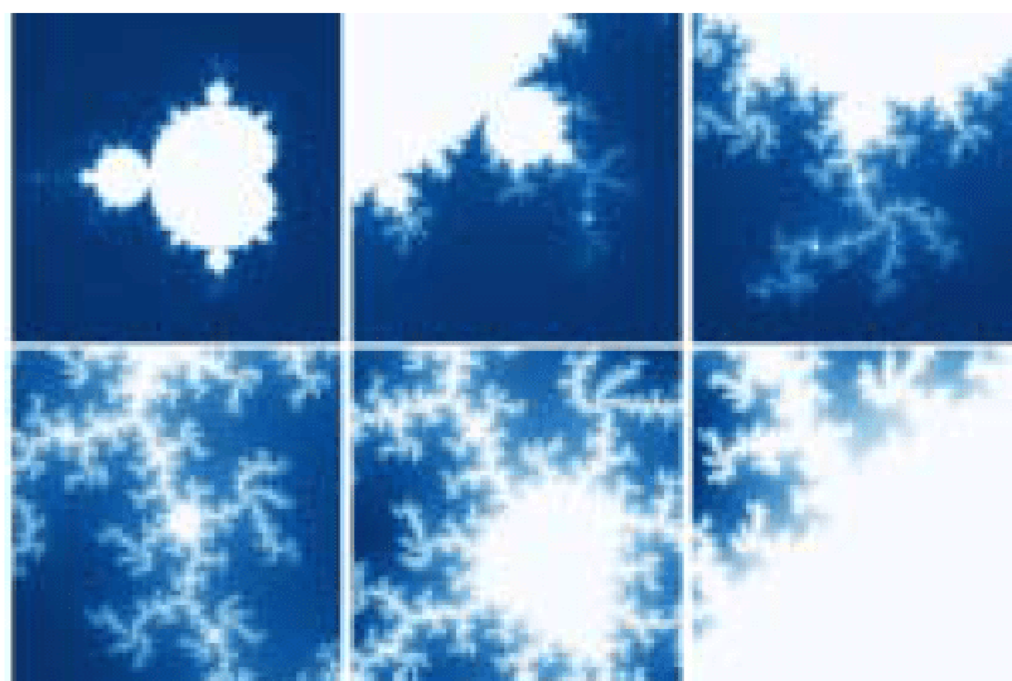


图 19-1　Mandelbrot 集合，以 5 倍倍率放大点(0.273, 0.595)附近

❶iter_point()计算点c的逃逸时间,逃逸半径R为2.0,最大迭代次数为100。❷draw_mandelbrot()绘制以点(cx, cy)为中心，边长为2*d的正方形区域内的Mandelbrot集合的图案。

❸计算指定范围内的参数c，它是一个二维的复数数组，形状为(200, 200)。这里用ogrid对象快速产生实部和虚部网格x和y，然后通过广播运算得到数组c。

❹接下来通过frompyfunc()将iter_point()转换为ufunc函数，这样便可以自动对c中的每个元素调用iter_point()进行运算。由于结果数组的元素类型为object，因此还需要调用astype()将其元素类型转换为浮点数类型。❺最后调用matplotlib的imshow()将结果数组绘制成图，通过cmap参数指定颜色映射表。

使用Python绘制Mandelbrot集合，最大的问题就是运算速度太慢，下面是图19-1中每幅图的计算时间：

```
time= 0.88162629608
time= 1.53712748408
time= 1.71502160191
time= 1.8691174437
time= 3.03812691278
```

因为每个点的逃逸时间均不相同，所以计算每幅图所需的时间也不相同。

19.1.1 使用NumPy加速计算

借助NumPy的数组运算可以提高计算速度。前面介绍的计算是在外层循环中对复平面上的每个点进行计算，在内部循环中计算每个点的逃逸时间。如果要用数组运算来加速计算，就需要将这两个循环的顺序颠倒。下面的程序演示了这一算法：



mandelbrot_numpy.py
用NumPy加速Mandelbrot集合计算

```
import numpy as np
import pylab as pl
import time
from matplotlib import cm

def draw_mandelbrot(cx, cy, d, N=200):
    """
    绘制点(cx, cy)附近在正负d范围之内的Mandelbrot
    """
    global mandelbrot

    x0, x1, y0, y1 = cx-d, cx+d, cy-d, cy+d
    y, x = np.ogrid[y0:y1:N*1j, x0:x1:N*1j]
    c = x + y*1j
```

```
    # 创建X、Y轴的坐标数组
    ix, iy = np.mgrid[0:N,0:N]

    # 创建保存mandelbrot图的二维数组，默认值为最大迭代次数
    mandelbrot = np.ones(c.shape, dtype=np.int)*100

    # 将数组都变成一维的
    ix.shape = -1
    iy.shape = -1
    c.shape = -1
    z = c.copy() # 从c开始迭代，因此开始的迭代次数为1

    start = time.clock()

    for i in xrange(1,100):
        # 进行一次迭代
        z *= z ❶
        z += c
        # 找到所有结果逃逸了的点
        tmp = np.abs(z) > 2.0 ❷
        # 将这些逃逸点的迭代次数赋值给mandelbrot图
        mandelbrot[ix[tmp], iy[tmp]] = i

        # 找到所有没有逃逸的点
        np.logical_not(tmp, tmp) ❸
        # 更新ix、iy、c、z，只包含没有逃逸的点
        ix,iy,c,z = ix[tmp], iy[tmp], c[tmp],z[tmp]
        if len(z) == 0: break

    print "time=",time.clock() - start
    pl.imshow(mandelbrot, cmap=cm.Blues_r, extent=[x0,x1,y0,y1])
    pl.gca().set_axis_off()

x,y = 0.27322626, 0.595153338

pl.subplot(231)
draw_mandelbrot(-0.5,0,1.5)
for i in range(2,7):
    pl.subplot(230+i)
    draw_mandelbrot(x, y, 0.2**(i-1))
pl.subplots_adjust(0.02, 0, 0.98, 1, 0.02, 0)
pl.show()
```

为了减少计算次数，程序中每次迭代之后，都将已经逃逸的点剔除出去，这样就需要保存每个点的下标。程序中用两个数组 ix 和 iy 保存没有逃逸的点的下标，因为有额外的数组保存下标，因此数组 z 和 c 不需要是二维的。

❶进行函数的迭代计算。使用“*=”、“+=”这样的运算符能够让 NumPy 不分配额外的空间，直接在数组 z 上进行运算。

❷计算逃逸点，此处的逃逸半径为 2。tmp 是逃逸点在 z 中的下标，由于 z、ix、iy 等数组始终是同时更新的，因此(ix[tmp], iy[tmp])就是逃逸点在图像中的下标。将数组 mandelbrot 中逃逸点的值设置为当前的迭代次数。

❸最后通过对 tmp 中的每个元素取逻辑反，更新所有没有逃逸的点对应的 ix、iy、c、z。

此程序的计算时间如下：

```
time= 0.186070576008
time= 0.327006365334
time= 0.372756034636
time= 0.410074464771
time= 0.681048289658
time= 0.878626752841
```

19.1.2 使用 Weave 加速计算

计算速度慢的最大原因在于 iter_point()中的循环运算。如果将此函数用 C 语言重写，将能显著提高计算速度。下面使用 SciPy 的 Weave 模块，将用 C++重写的 iter_point()嵌入到 Python 程序中：



mandelbrot_weave.py
用 Weave 加速 Mandelbrot 集合的计算

```
def weave_iter_point(c):
    code = """
    std::complex<double> z;
    int i;
    z = c;
    for(i=1;i<100;i++)
    {
        if(std::abs(z) > 2) break;
        z = z*z+c;
    }
    return_val=i;
    """

    f = weave.inline(code, ["c"], compiler="gcc")
    return f
```

下面是使用 weave_iter_point()计算 Mandelbrot 集合的时间：

```
time= 0.285266982256
time= 0.271430028118
time= 0.293769180161
time= 0.308515188383
time= 0.411168179196
```

19.1.3　连续的逃逸时间

修改逃逸半径 R 和最大迭代次数 N，可以绘制出不同效果的 Mandelbrot 集合图案。但是使用前面所讲方法计算出的逃逸时间大于逃逸半径时的迭代次数，因此输出的图像最多只有 N 种不同的颜色值，有很强的梯度感。为了在不同的梯度之间进行渐变处理，可是使用下面的公式对逃逸时间进行计算：

$$n - \log_2 \log_2 |Z_n|$$

Z_n 是迭代 n 次之后的结果，通过在逃逸时间的计算中引入迭代结果的模值，结果将不再是整数，而是平滑渐变的。下面是计算逃逸时间的程序：



mandelbrot_smooth_python.py
使用逃逸时间平滑 Mandelbrot 集合图案

```
def smooth_iter_point(c):
    z = c
    for i in xrange(1, iter_num):
        if abs(z)>escape_radius: break
        z = z*z+c
    absz = abs(z)
    if absz > 2.0:
        mu = i - log(log(absz,2),2)
    else:
        mu = i
    return mu # 返回正规化的迭代次数
```

如果逃逸半径设置得很小，例如 2.0，那么有可能结果不够平滑。这时可以在迭代循环之后添加几次迭代，以保证 z 的模值足够大。例如在迭代循环之后添加如下代码：

```
z = z*z+c
z = z*z+c
i += 2
```

图 19-2 是逃逸半径为 10、最大迭代次数为 20 的结果(见封三彩插)。

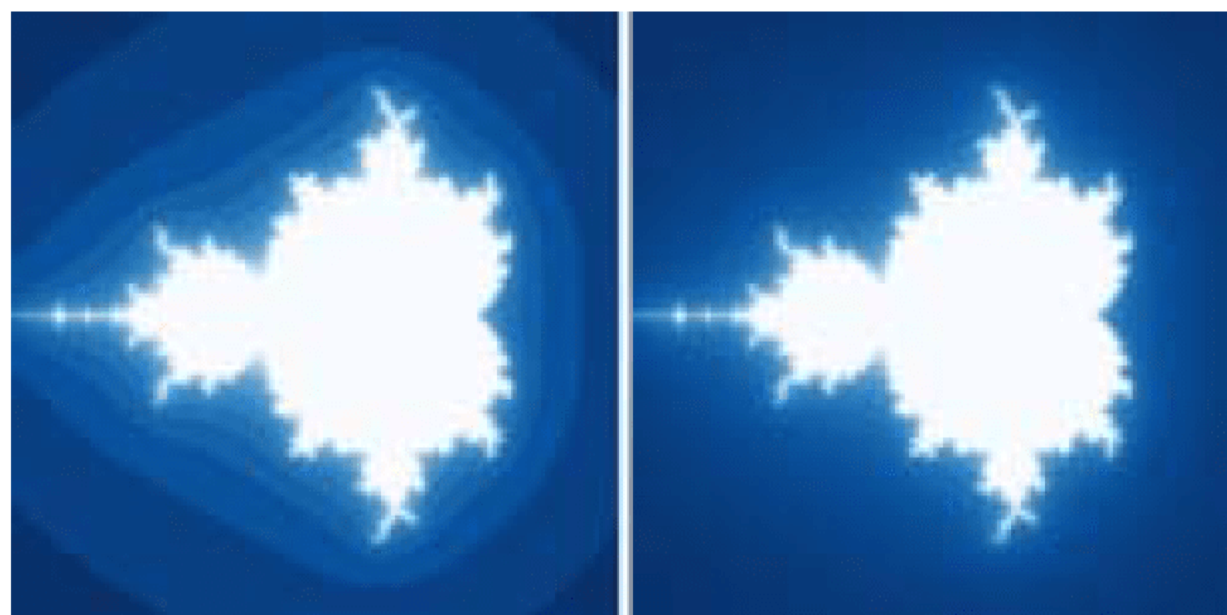


图 19-2　平滑处理后的 Mandelbrot 集合：逃逸半径=10，最大迭代次数=20



http://linas.org/art-gallery/escape/ray.html
逃逸时间公式的详细推导

19.1.4　Mandelbrot 演示程序

为了实时计算 Mandelbrot 集合的图像，我们需要更快的运算速度，可以将所有的循环计算逃逸时间的部分都使用 C 语言编写，然后用 Weave 模块将其嵌入到 Python 程序中。下面是用 Chaco 制作的实时绘制 Mandelbrot 集合的演示程序，界面截图如图 19-3 所示(见封三彩插)。



mandelbrot_weave_demo.py
实时绘制 Mandelbrot 集合的演示程序

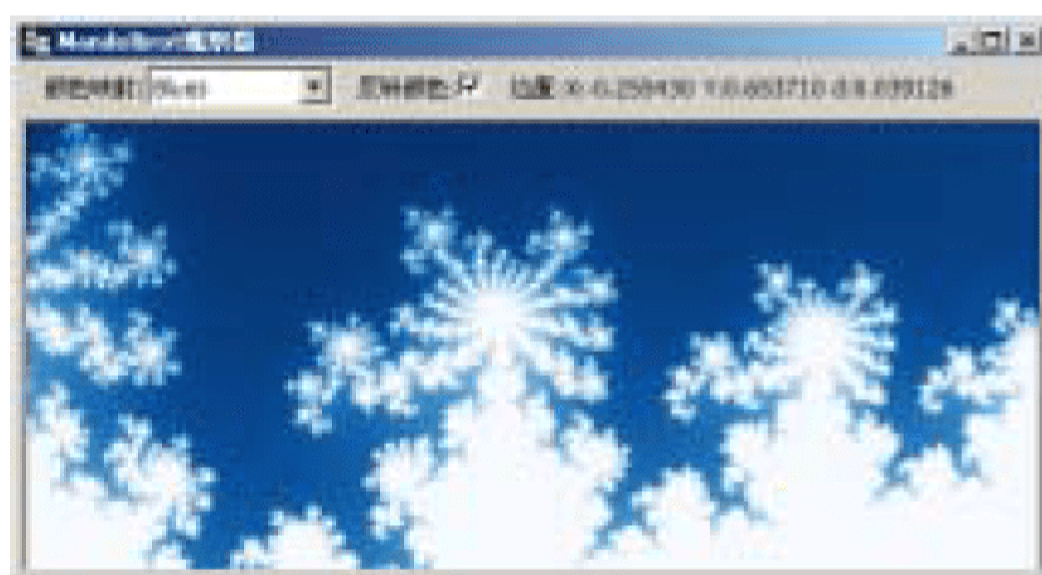


图 19-3　实时绘制 Mandelbrot 集合的演示程序

在界面中，用户可以通过鼠标滚轴对 Mandelbrot 图像进行缩放，使用鼠标左键拖动图像的显示范围，并且可以用图像上方的工具条修改图像的颜色映射表。

下面是计算 Mandelbrot 集合图像的函数。参数 cx 和 cy 是复平面上计算范围的中心点，参数 d 是中心点到计算边界的实轴上的长度，参数 arr 是保存计算结果的二维数组：

```
def weave_mandelbrot(cx, cy, d, arr):
    cx = float(cx)
```

```
        cy = float(cy)
        d = float(d)
        code = """
        double x, y;
        int h = Narr[0];
        int w = Narr[1];
        double dx = 2*d / (double)w;
        y = cy-d*(double)h/w;
        for(int i=0;i<h;i++)
        {
            x = cx-d;
            for(int j=0;j<w;j++)
            {
                int k;
                double zx, zy;
                zx = x; zy = y;
                for(k=1;k<100;k++)
                {
                    double tx, ty;
                    if(zx*zx + zy*zy > 4) break;
                    tx = zx*zx - zy*zy + x;
                    ty = 2*zx*zy + y;
                    zx = tx;
                    zy = ty;
                }
                double absz = sqrt(zx*zx + zy*zy);
                if(absz>2.0) arr(i,j) = k - log(log(absz)/log(2))/log(2);
                else arr(i,j) = k;
                x += dx;
            }
            y += dx;
        }
        """
        weave.inline(code, ["cx", "cy", "d", "arr"], compiler="gcc",
                    type_converters=converters.blitz)
```

19.2 迭代函数系统(IFS)

迭代函数系统是一种创建分形图案的简单算法，它所创建的分形图案永远是绝对自相似的。下面我们以绘制某种蕨类植物叶子的图案为例，介绍迭代函数系统算法以及如何用 Python 来实现。

用下面 4 个线性函数对二维平面上的坐标进行线性变换：

```
1.
    x(n+1)= 0
    y(n+1) = 0.16 * y(n)

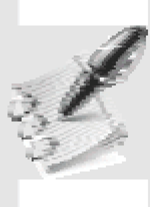
2.
    x(n+1) = 0.2 * x(n) - 0.26 * y(n)
    y(n+1) = 0.23 * x(n) + 0.22 * y(n) + 1.6

3.
    x(n+1) = -0.15 * x(n) + 0.28 * y(n)
    y(n+1) = 0.26 * x(n) + 0.24 * y(n) + 0.44

4.
    x(n+1) = 0.85 * x(n) + 0.04 * y(n)
    y(n+1) = -0.04 * x(n) + 0.85 * y(n) + 1.6
```

所谓迭代函数系统，是指将函数的输出再次当做输入进行迭代计算，即用上面公式将坐标 x(n)、y(n)变换成坐标 x(n+1)、y(n+1)。问题是有 4 个迭代函数，迭代时选择哪个函数进行计算呢？我们为每个函数指定一个概率值，它们依次为 1%、7%、7%、85%。通过每个函数的概率随机选择一个函数进行迭代。上面的例子中，第 4 个函数被选择进行迭代的概率最高。

最后，我们从坐标原点(0,0)开始迭代，绘制每次迭代得到的坐标点，进而得到迭代函数系统的分形图案。下面的程序演示了这一计算过程：



IFS_fern.py
使用迭代函数系统绘制叶子的分形图案

```
import numpy as np
import matplotlib.pyplot as pl
import time

# 蕨类植物叶子的迭代函数及其概率值
eq1 = np.array([[0,0,0],[0,0.16,0]])
p1 = 0.01

eq2 = np.array([[0.2,-0.26,0],[0.23,0.22,1.6]])
p2 = 0.07

eq3 = np.array([[-0.15, 0.28, 0],[0.26,0.24,0.44]])
p3 = 0.07

eq4 = np.array([[0.85, 0.04, 0],[-0.04, 0.85, 1.6]])
p4 = 0.85
```

```
def ifs(p, eq, init, n):
    """
    进行函数迭代
    p: 每个函数的选择概率列表
    eq: 迭代函数列表
    init: 迭代初始点
    n: 迭代次数

    返回值：每次迭代所得的 X 坐标数组，Y 坐标数组，计算所用的函数下标
    """

    # 迭代向量的初始化
    pos = np.ones(3, dtype=np.float) ❶
    pos[:2] = init

    # 通过函数概率，计算函数的选择序列
    p = np.add.accumulate(p)        ❷
    rands = np.random.rand(n)
    select = np.ones(n, dtype=np.int)*(n-1)
    for i, x in enumerate(p[::-1]):
        select[rands<x] = len(p)-i-1

    # 结果的初始化
    result = np.zeros((n,2), dtype=np.float)
    c = np.zeros(n, dtype=np.float)

    for i in xrange(n):
        eqidx = select[i] # 所选的函数下标
        tmp = np.dot(eq[eqidx], pos) # 进行迭代
        pos[:2] = tmp # 更新迭代向量

        # 保存结果
        result[i] = tmp
        c[i] = eqidx

    return result[:,0], result[:, 1], c

start = time.clock()
x, y, c = ifs([p1,p2,p3,p4],[eq1,eq2,eq3,eq4], [0,0], 100000)
print time.clock() - start
pl.figure(figsize=(6,6))
pl.subplot(121)
pl.scatter(x, y, s=1, c="g", marker="s", linewidths=0) ❸
pl.axis("equal")
pl.axis("off")
pl.subplot(122)
```

```
    pl.scatter(x, y, s=1, c = c, marker="s", linewidths=0) ❹
    pl.axis("equal")
    pl.axis("off")
    pl.subplots_adjust(left=0,right=1,bottom=0,top=1,wspace=0,hspace=0)
    pl.gcf().patch.set_facecolor("white")
    pl.show()
```

ifs()是进行函数迭代的主函数。❶我们希望通过矩阵乘法 dot()计算坐标变换的结果，因此需要将初始迭代坐标扩展为三维齐次坐标。这样一来，与迭代系数进行矩阵乘积运算的向量就变成了(x(n), y(n), 1.0)。

❷为了减少计算时间，我们不在迭代循环中计算选择迭代方程的随机数，而是事先通过每个迭代方程的概率，计算出选择数组 select。注意这里使用 accumulate()先将概率累加，然后产生一组 0 到 1 之间的随机数，通过判断随机数所在的区间来选择不同的方程下标。也可以使用 SciPy 的 stats 模块中的离散随机变量来产生这个随机下标数组。

❸最后通过 scatter()将得到的坐标点绘制成散列图。其中：s 参数是每个散列点的大小，因为我们要绘制 10 万个点，为了提高绘图速度，我们选择点的大小为 1 个像素；c 参数是点的颜色，这里选择绿色；marker 参数是点的形状，"s"表示正方形，方形的绘制是最快的；linewidths 参数是点的边框宽度，0 表示没有边框。

❹此外，c 参数还可以传入一个数组，作为每个点的颜色值。我们将计算坐标的公式下标传入，这样可以直观地看出点是由哪个公式迭代产生的。

图 19-4 是程序的输出结果(见封三彩插)。观察右图中 4 种颜色的分布，可以发现概率为 1%的迭代方程所计算的是叶杆部分(深蓝色)；概率为 7%的两个迭代方程计算的是左右两片子叶；而概率为 85%的迭代方程计算整个叶子，最下面的三种颜色的点通过此公式迭代产生上面所有的点。可以看出整片叶子呈现出完美的自相似特性，任意取其中的一片子叶，将其旋转放大之后，和整片叶子完全相同。



图 19-4　迭代函数系统所绘制的蕨类植物的叶子

19.2.1 二维仿射变换

上一节介绍的 4 个坐标变换方程的一般形式如下：

```
x(n+1) = A * x(n) + B * y(n) + C
y(n+1) = D * x(n) + E * y(n) + F
```

这种变换被称为二维仿射变换，它是将二维坐标变换到其他二维坐标的线性映射，并保留直线性和平行性。即直线经过变换之后仍然是一条直线，原来平行的直线，变换之后仍然是平行的。这种变换可以看成是由一系列平移、缩放、旋转变换构成的。

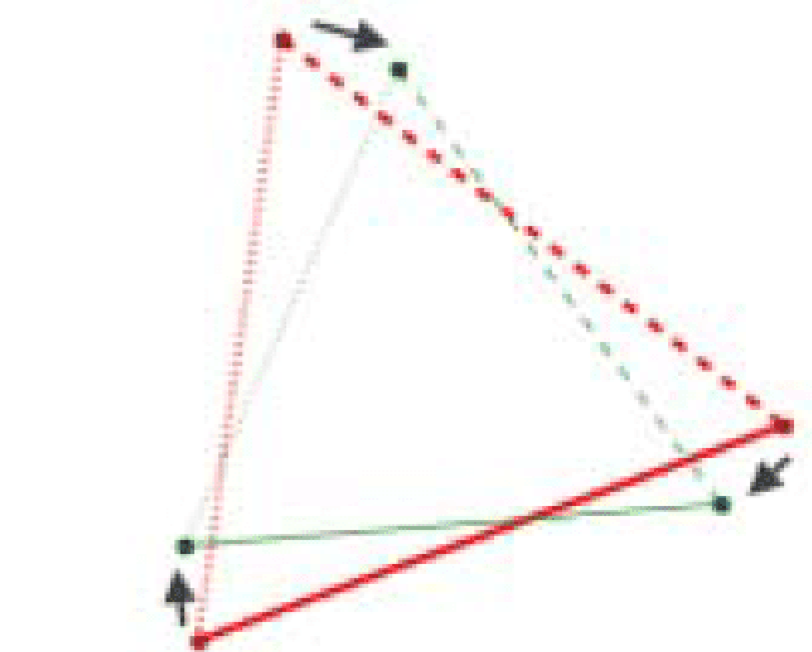


图 19-5 两个三角形决定二维仿射变换的 6 个参数

为了直观地显示仿射变换，我们可以使用平面上的两个三角形来表示。因为仿射变换公式中有 6 个未知数——A、B、C、D、E、F，而每两个点之间的变换可以决定两个方程，因此一共需要三组点来决定 6 个变换方程，正好是两个三角形，如图 19-5 所示。

图中，从红色三角形的每个顶点变换到绿色三角形的对应顶点，正好能够决定仿射变换中的 6 个参数。这样的话，我们可以使用 $N+1$ 个三角形，决定 N 个仿射变换，其中每个变换的参数都是由第 0 个三角形和其他的三角形决定的。第 0 个三角形我们称为基础三角形，其他的三角形称为变换三角形。

为了绘制迭代函数系统的图像，我们还需要给每个仿射变换方程指定一个迭代概率参数。此参数也可以使用三角形直观地表达出来：迭代概率和变换三角形的面积成正比，即迭代概率为变换三角形的面积除以所有变换三角形的面积之和。

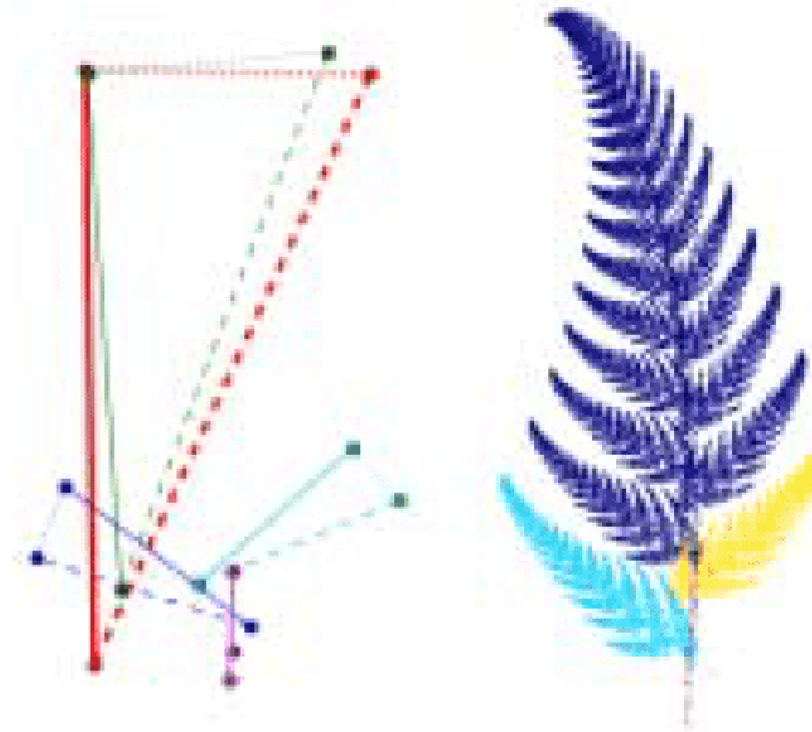


图 19-6 5 个三角形的仿射变换方程绘制蕨类植物的叶子

如图 19-6 所示(见封三彩插)，前面介绍的蕨类植物的分形图案的迭代方程可以由 5 个三角形决定，可以很直观地看出紫色小三角形决定了叶子的茎；而两个蓝色的三角形决定了左右两片子叶；绿色的三角形将茎和两片子叶往上复制，形成整片叶子。

19.2.2 迭代函数系统设计器

按照前面介绍的三角形法，我们可以设计一个用于迭代函数系统的设计工具。用户通过程序界面绘制或修改一组三角形，程序计算这组三角形所对应的迭代方程组的系数，并实时地绘

制迭代图案。本书提供了 matplotlib 和 Chaco 两个版本的迭代函数设计程序，读者可以通过分析它们的源程序来了解如何制作交互式绘图程序。图 19-7 是使用 matplotlib 制作的界面截图(见封三彩插)，而图 19-8 是使用 Chaco 制作的界面截图(见封三彩插)。



ifs_matplotlib.py
使用 matplotlib 制作的迭代函数系统设计工具



ifs_chaco.py
使用 Chaco 制作的迭代函数系统设计工具

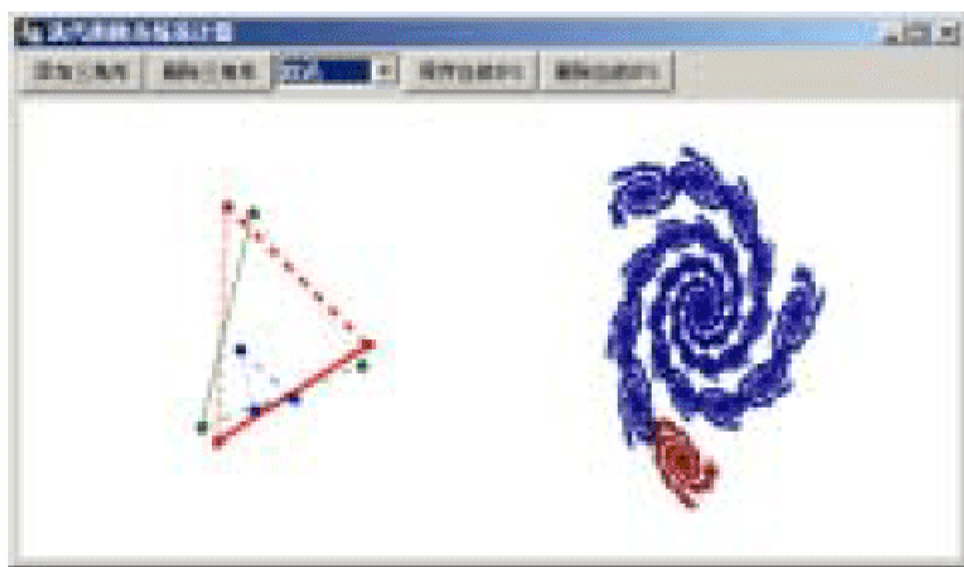


图 19-7 使用 matplotlib 制作的迭代函数系统分形图案设计界面

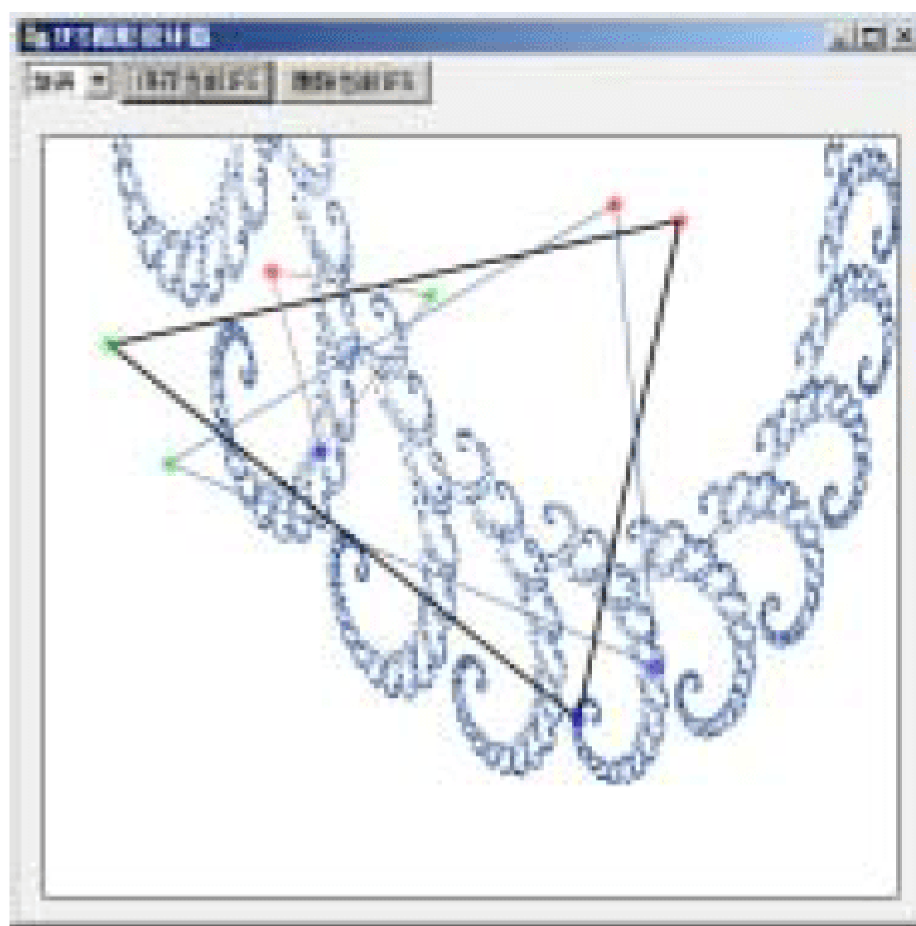


图 19-8 使用 Chaco 制作的迭代函数系统分形图案设计界面

这两个程序都通过调用“ifs_library.py”模块中的函数来完成函数的迭代计算。



ifs_library.py
迭代函数系统的计算模块

通过两个三角形求解仿射变换方程的系数，相当于求 6 元线性方程组的解，这个计算通过 solve_eq()函数完成，它先计算出线性方程组的矩阵 a 和 b，然后调用 NumPy 的 linalg.solve()对线性方程组 a × X = b 求解：

```
def solve_eq(triangle1, triangle2):
    """
    解方程，从 triangle1 变换到 triangle2 的变换系数
        triangle1、triangle2 是二维数组：
        x0,y0
```

```
        x1,y1
        x2,y2
    """
    x0,y0 = triangle1[0]
    x1,y1 = triangle1[1]
    x2,y2 = triangle1[2]

    a = np.zeros((6,6), dtype=np.float)
    b = triangle2.reshape(-1)
    a[0, 0:3] = x0,y0,1
    a[1, 3:6] = x0,y0,1
    a[2, 0:3] = x1,y1,1
    a[3, 3:6] = x1,y1,1
    a[4, 0:3] = x2,y2,1
    a[5, 3:6] = x2,y2,1

    c = np.linalg.solve(a, b)
    c.shape = (2,3)
    return c
```


我们使用三角形的面积计算每个仿射变换方程的迭代概率，面积通过 triangle_area()计算，它使用 NumPy 的 cross()来计算三角形两个边的矢量的叉积：

```
def triangle_area(triangle):
    """
    计算三角形的面积
    """
    A, B, C = triangle
    AB = A-B
    AC = A-C
    return np.abs(np.cross(AB,AC))/2.0
```

在两个界面程序中，都使用定时器周期性地调用进行迭代计算的函数。由于定时器所调用的函数在界面线程中运行，如果它的处理时间过长，将会影响界面的响应。因此迭代函数每次只计算 ITER_COUNT 个点。此外，如果计算的总点数过多，会影响显示速度，因此最多只调用 ITER_TIMES 次迭代函数。这两个参数可以在程序的开头找到，请读者根据自己计算机的运算速度进行修改，下面是“ifs_chaco.py”中的默认设置：

```
ITER_COUNT = 2000 # 一次 IFS 迭代的点数
ITER_TIMES = 20   # 总共调用 IFS 的次数
```

19.3 L-System 分形

前面绘制的分形图案都是使用数学函数迭代产生的，而 L-System 分形则是采用符号迭代产生的。首先定义如下几个有含义的符号：

- F：向前走固定单位。
- +：正方向旋转固定单位。
- -：负方向旋转固定单位。

使用这三个符号，我们很容易描述图 19-9(左上)中由 4 条线段构成的图案：

```
F+F--F+F
```

如果将此符号串中的所有 F 都替换为 F+F--F+F，就能得到如下的新字符串：

```
F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F
```

如此替换迭代下去，并根据字符串进行绘图(符号+和-分别表示正负旋转 60°)，可得到如图 19-9 所示的分形图案：

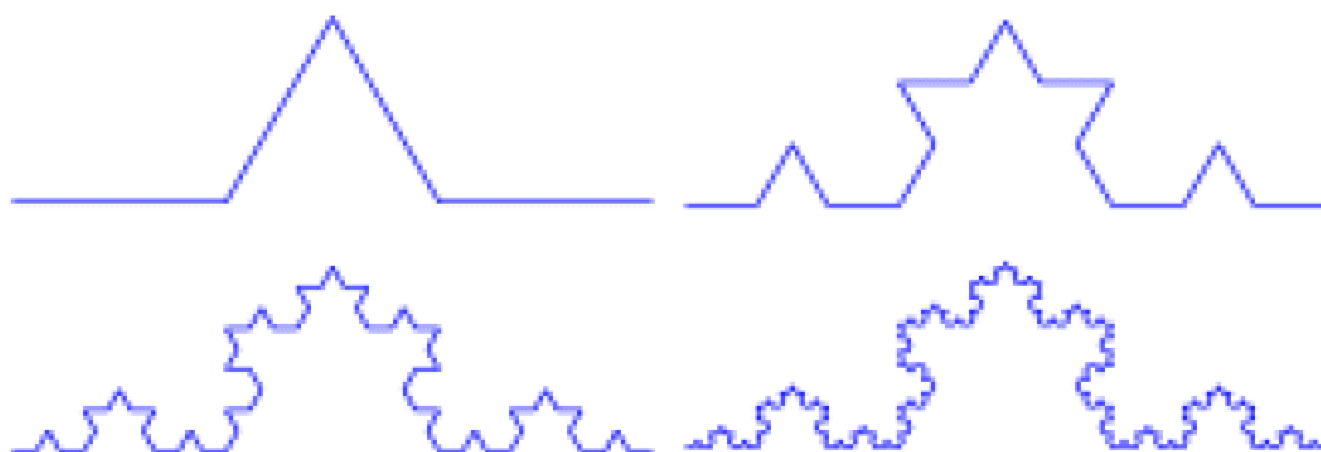


图 19-9　使用 F+F--F+F 进行迭代的分形图案

除了“F”、“+”、“-”之外，我们再定义如下几个符号：

- f：向前走固定单位，它和 F 的效果相同，但可以用它定义不同的迭代公式。
- [：将当前的位置压入堆栈。
-]：从堆栈中读取坐标，修改当前位置。
- S：初始迭代符号，所有的迭代从此符号开始。

所有的符号(包括上面未定义的)都可以用来定义迭代，通过引入堆栈，我们能描述带分叉的图案。例如下面的符号迭代能够绘制出一棵植物：

```
S -> X
X -> F-[[X]+X]+F[+FX]-X
F -> FF
```

我们用一个字典定义所有的迭代公式和其他的一些绘图信息：

```
{
    "X":"F-[[X]+X]+F[+FX]-X", "F":"FF", "S":"X",
    "direct":-45,
    "angle":25,
    "iter":6,
    "title":"Plant"
}
```

其中：direct 是绘图的初始角度，通过指定不同的值可以旋转整个图案；angle 是符号“+”和“-”的旋转角度，不同的值能产生完全不同的图案；iter 是迭代次数。

下面的程序实现 L-System 的迭代工作，并计算每段线段的坐标。



l_system.py
绘制 L-System 分形图案

```
class L_System(object):
    def __init__(self, rule):
        info = rule['S']
        for i in range(rule['iter']):
            ninfo = []
            for c in info:
                if c in rule:
                    ninfo.append(rule[c])
                else:
                    ninfo.append(c)
            info = "".join(ninfo)
        self.rule = rule
        self.info = info

    def get_lines(self):
        d = self.rule['direct']
        a = self.rule['angle']
        p = (0.0, 0.0)
        l = 1.0
        lines = []
        stack = []
        for c in self.info:
            if c in "Ff":
                r = d * pi / 180
                t = p[0] + l*cos(r), p[1] + l*sin(r)
                lines.append(((p[0], p[1]), (t[0], t[1])))
                p = t
```

```
            elif c == "+":
                d += a
            elif c == "-":
                d -= a
            elif c == "[":
                stack.append((p,d))
            elif c == "]":
                p, d = stack[-1]
                del stack[-1]
        return lines
```

下面的 draw()完成绘图工作：

```
from matplotlib import collections
def draw(ax, rule, iter=None):
    if iter!=None:
        rule["iter"] = iter
    lines = L_System(rule).get_lines() ❶
    linecollections = collections.LineCollection(lines) ❷
    ax.add_collection(linecollections, autolim=True) ❸
    ax.axis("equal")
    ax.set_axis_off()
    ax.set_xlim(ax.dataLim.xmin, ax.dataLim.xmax)
    ax.invert_yaxis()
```

❶get_lines()返回每个线段的坐标。❷创建一个表示所有线段集合的 LineCollection 对象，❸并调用 Axes 对象 ax 的 add_collection()将此线段集合添加到 ax.collections 列表中。这样能一次添加多条线段，提高绘图速度。图 19-10 是程序绘制的几种 L-System 分形图案。

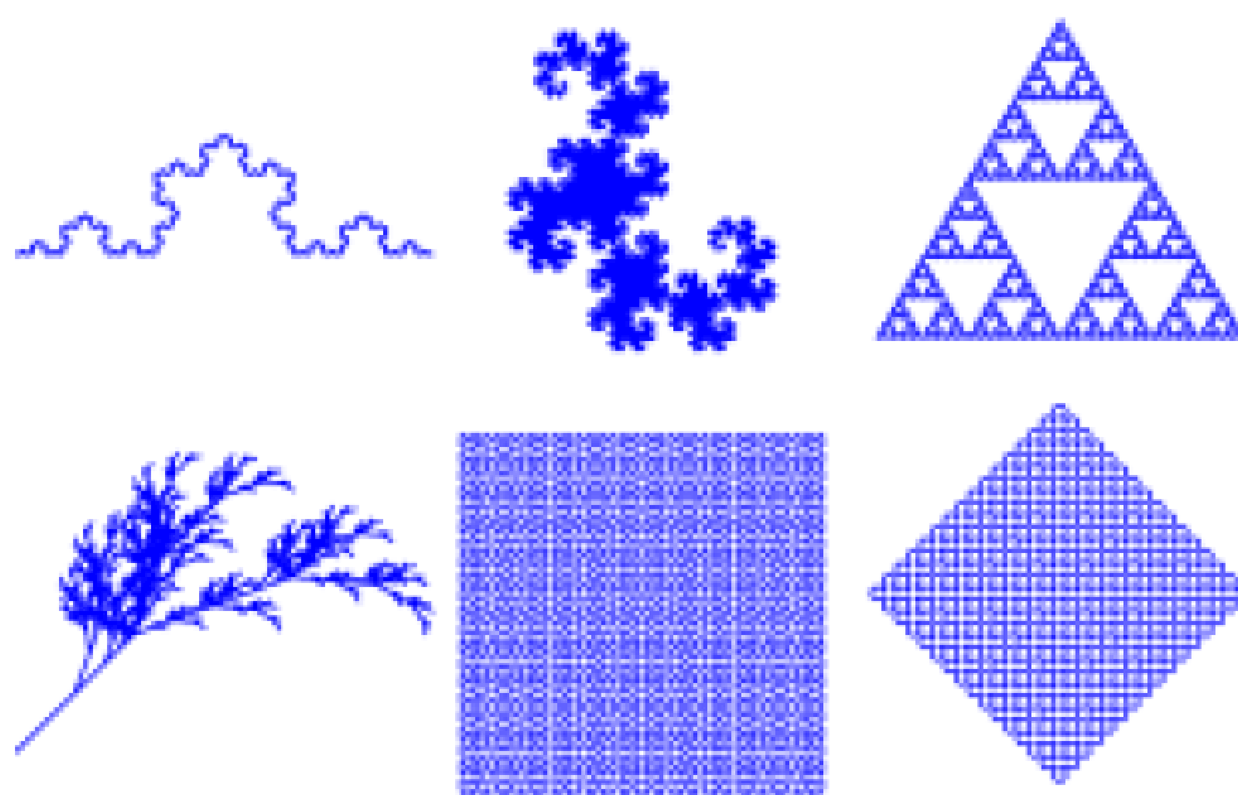


图 19-10　几种 L-System 分形图案

19.4 分 形 山 脉

前面介绍的分形图案都是严格按照迭代公式计算出来的，然而自然界中的山川、云彩、树木等都不是精确的自相似图形，而是统计意义上的自相似图形。本节将介绍几种经典的随机山脉地形的生成算法，以及如何用 Python 快速实现这些算法。

19.4.1 一维中点移位法

让我们从绘制一条分形曲线开始。使用中点位移算法(midpoint displacement)，能够有效地模拟山脉或海岸线的分形形状，算法如下：

(1) 首先在 X 轴上取两个初始点 A 和 B。

(2) 找到 AB 两点的中点，并在 Y 轴方向上进行随机移位，移位后的点为 C。

(3) 对于线段 AC 和 BC 进行与步骤(2)相同的操作。

每次迭代时，随机移位的最大幅度都成比例地衰减。迭代足够多次之后，将得到的点连接起来，就得到了一条随机的分形曲线。

下面是实现此算法的源程序，程序绘制的山脉曲线如图 19-11 所示(见封三彩插)。



fractal_hill1D.py
使用中点移位法绘制一维分形山脉

```
def hill1d(n, d):
    """
    绘制山脉曲线，2**n+1 为曲线在 X 轴上的长度，d 为衰减系数
    """
    a = np.zeros(2**n+1) ❶
    scale = 1.0
    for i in xrange(n, 0, -1): ❷
        s = 2**(i-1) ❸
        s2 = 2*s
        tmp = a[::s2]
        a[s::s2] += (tmp[:-1] + tmp[1:]) * 0.5 ❹
        a[s::s2] += np.random.normal(size=len(tmp)-1, scale=scale) ❺
        scale *= d ❻
    return a

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    plt.figure(figsize=(8,4))
    for i, d in enumerate([0.4, 0.5, 0.6]):
        np.random.seed(8) ❼
```

```
        a = hill1d(9, d)
        plt.plot(a, label="d=%s" % d, linewidth=3-i)
    plt.xlim((0,len(a)))
    plt.legend()
    plt.show()
```

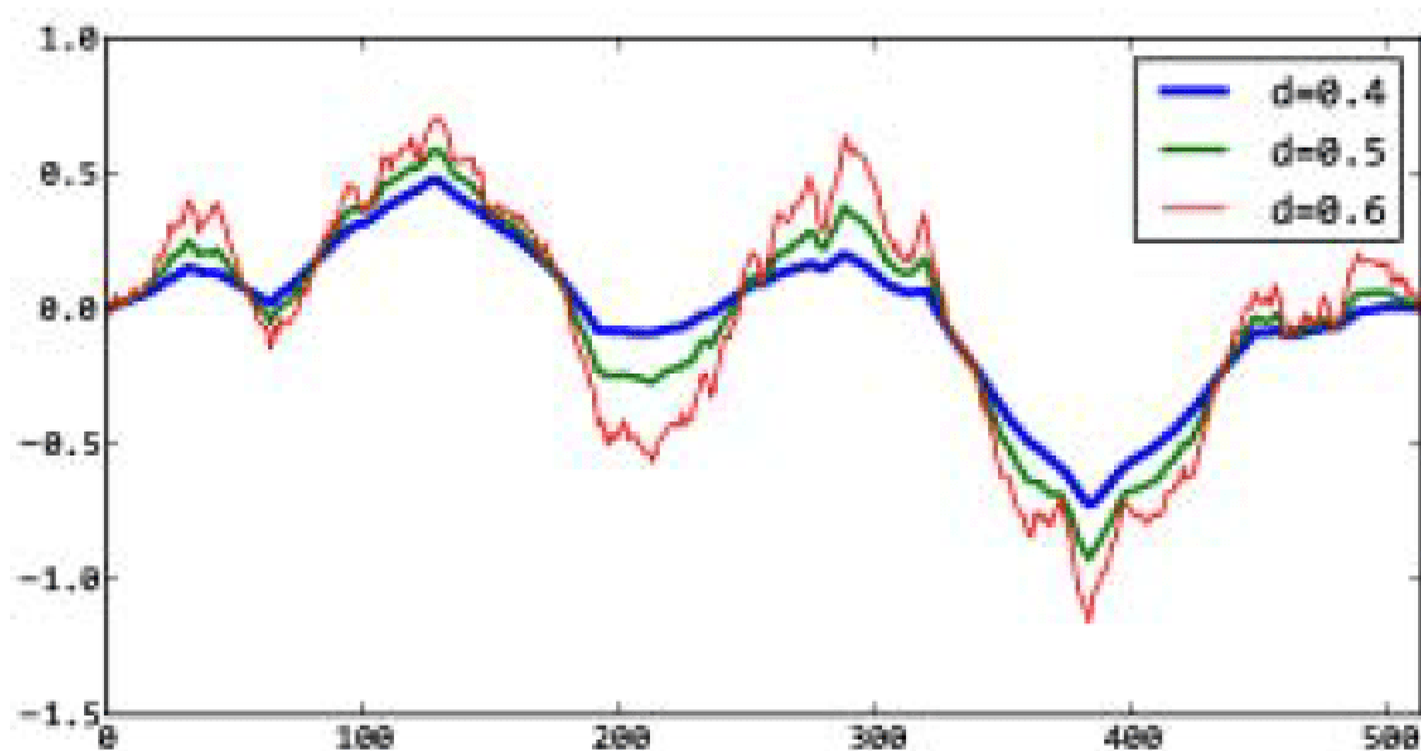


图 19-11　一维分形山脉曲线，衰减值越小，最大幅度的衰减越快，曲线越平滑

hill1d()计算一维分形山脉曲线。❶为了运算方便，我们用一个一维数组 a 保存曲线上每点的高度，而曲线上每点的 X 轴坐标则由数组的下标决定。这要求每次计算中点时得到的 X 轴坐标必须是整数。显然，当数组长度为 2^n+1 时满足这个要求。

❷由于数组 a 的长度为 2^n+1，因此需要循环 n 次才能够计算到数组上所有的点。每次循环时，都要对数组中的某些点计算中值，例如对于 n=8，即数组长度为 257 时，循环变量 i 和数组中需要计算中点的下标如表 19-1 所示。

表 19-1　数组长度为 257 时需要计算中点的下标

i	需要计算中点的数组下标
8	128
7	64,192
6	32,96,160,224
5	16, 48, 80, 112, 144, 176, 208, 240

❸数组中每次迭代要计算的中点下标是一个等差数列，其起始下标为 $s=2^{i-1}$，间隔为 $s=2^i$。❹而每个中点都由其左右下标相差 s 的两个数值计算。❺给每个中点一定的随机位移。这里使用 normal()产生一个正态分布的随机数组，其期望值为 0，标准偏差为 scale。这样产生的山脉曲线就会既有山峰也有山谷。❻最后将下一次迭代的标准偏差乘上系数 d，d 越小，标准偏差的衰减越快。

接下来绘制 d 为 0.4、0.5、0.6 时的山脉曲线。❼这里为了对不同的衰减系数所产生的曲线进行比较，需要保证每次都使用相同的随机数来计算曲线，因此使用 seed()指定生成随机数的

种子。在真正需要随机产生曲线时，请将此句注释掉。

19.4.2　二维中点移位法

中点移位法很容易扩展到二维，我们可以使用它计算山脉曲面，算法如图 19-12 所示。左图中，从白色圆点的值(值为 0、2、4、8 的 4 个点)计算它们的 5 个用灰色圆点表示的中值点。边上 4 个中值点的计算和一维的情况相同，而正中间的中值则是 4 个角上的值的平均值。

右图以计算 5*5 的方格为例，演示了每步迭代时所计算的点。方格中的数字表示计算此点的值的迭代次数。初期情况下 4 个角上的点已知，标记它们的迭代次数为 0。根据左图的中值计算方式，计算出标记为 1 的 5 个方格的值。然后对于由迭代 0 和迭代 1 的点组成的 4 个方块，再次进行中值计算，计算出所有标记为 2 的 16 个方格的值。

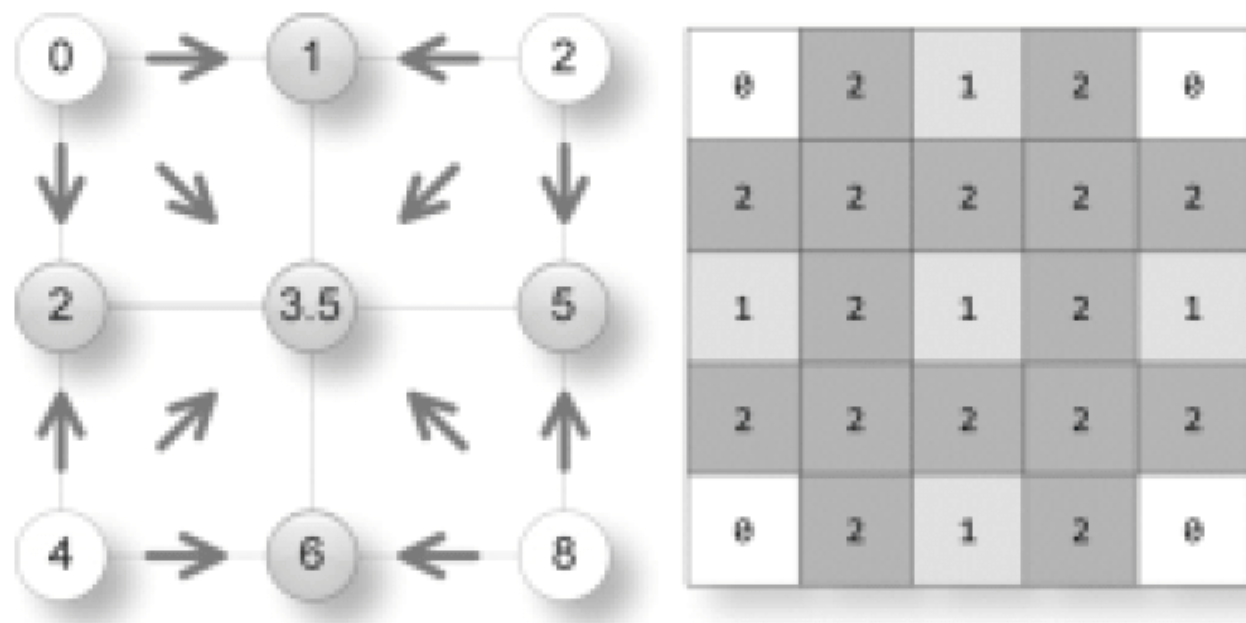


图 19-12　二维中点移位法示意图

完整的计算程序如下，程序的显示效果如图 19-13 所示(见封三彩插)。

fractal_hill2D.py
二维中点移位法绘制山脉曲面

程序的算法和一维的情况类似，这里就不多解释了。❶在计算出表示山脉曲面的二维数组 a 之后，调用 np.ptp()得到数组 a 中最大值和最小值之间的差，并将值放大到数组形状的 0.5 倍，以便调用 mlab.surf()绘制曲面。❷使用 SciPy 的多维数组卷积函数 convolve()对二维数组 a 进行平滑处理。

```
import numpy as np
import pylab as pl
from numpy.random import normal

def hill2d(n, d):
    """
    绘制山脉曲面，曲面是一个(2**n + 1)*(2**n + 1)的图像
    d 为衰减系数
```

```
    """
    size = 2**n + 1
    scale = 1.0
    a = np.zeros((size, size))

    for i in xrange(n, 0, -1):
        s = 2**(i-1)
        s2 = s*2
        tmp = a[::s2,::s2]
        tmp1 = (tmp[1:,:] + tmp[:-1,:])*0.5
        tmp2 = (tmp[:,1:] + tmp[:,:-1])*0.5
        tmp3 = (tmp1[:,1:] + tmp1[:,:-1])*0.5
        a[s::s2, ::s2] = tmp1 + normal(0,scale,tmp1.shape)
        a[::s2, s::s2] = tmp2 + normal(0,scale,tmp2.shape)
        a[s::s2,s::s2] = tmp3 + normal(0,scale,tmp3.shape)
        scale *= d

    return a

if __name__ == "__main__":
    from enthought.mayavi import mlab
    from scipy.ndimage.filters import convolve
    a = hill2d(8, 0.5)
    a/= np.ptp(a) / (0.5*2**8) ❶
    a = convolve(a, np.ones((3,3))/9) ❷
    mlab.surf(a)
    mlab.show()
```

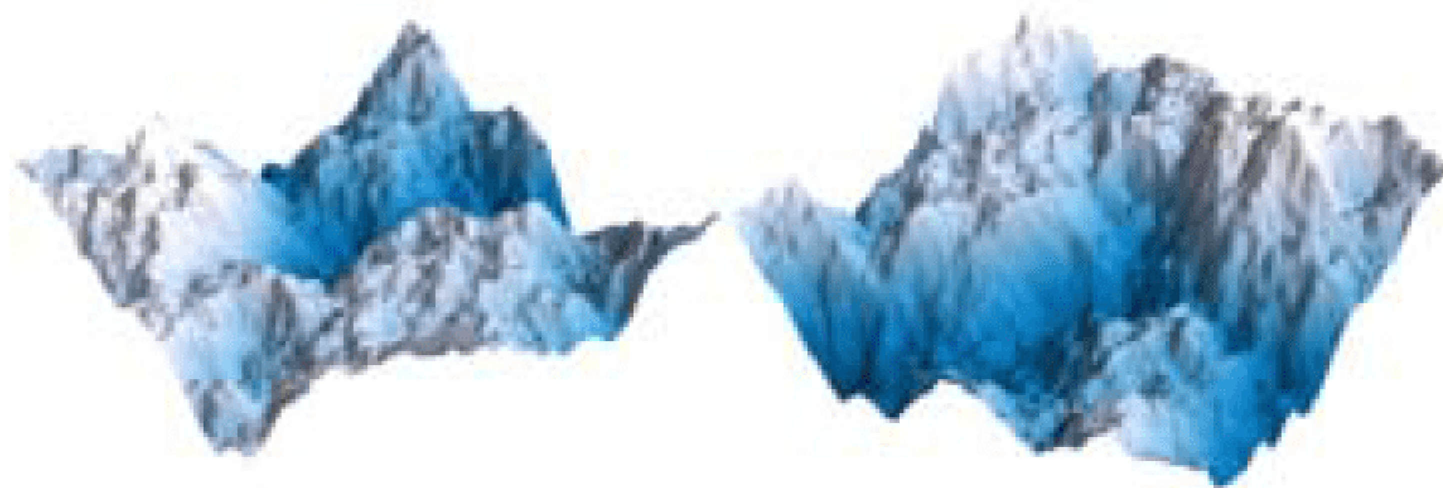


图 19-13　二维中点移位法绘制山脉曲面

19.4.3　菱形方形算法

通过上面的介绍我们知道，每次迭代都是通过正方形 4 个角上的点的值，计算边上 4 点和中心点的值，这是最简单的一种方法。但如果读者放大它所生成的曲面，就会发现曲面上有一些大大小小的正方形痕迹。菱形方形算法(diamond-square algorithm)通过将两种中值计算方法交替使用，有效地消除了这种正方形痕迹，算法如图 19-14 所示。

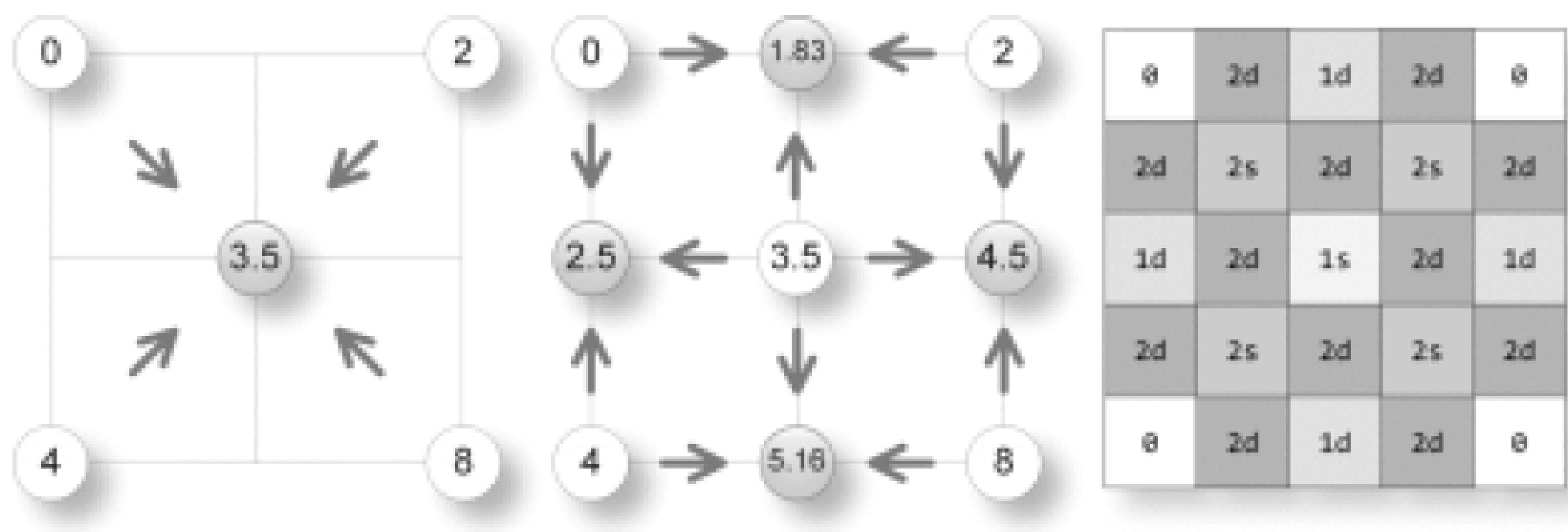


图 19-14　菱形方形算法

首先如左图所示，通过正方形的 4 个角点计算位于它们中心的点的平均值。然后如中图所示，通过菱形的 4 个角点计算位于它们中心的点的平均值。图中没有一个完整的菱形，而是 4 个半边菱形。我们也可以把正方形平均看做左上、右上、左下和右下 4 个方向上的点的平均值，而把菱形平均看做上、下、左、右 4 个方向上的点的平均值。右图显示了采用菱形正方形计算 5*5 的数组时的运算顺序。从标记为 0 的方格开始，以方形平均计算标记为 1s 的方格值，然后以菱形平均计算标记为 1d 的 4 个方格的值。接下来重复上面的步骤，方形平均计算 2s 方格，最后菱形平均计算 2d 方格。

使用菱形方形算法绘制山脉曲面的程序如下，hill2d_ds()是计算曲面数组的函数。

fractal_hill2D_ds.py
用菱形方形算法绘制山脉曲面

```
def hill2d_ds(n, d):
    size = 2**n + 1
    scale = 1.0
    a = np.zeros((size, size))

    for i in xrange(n, 0, -1):
        s = 2**(i-1)
        s2 = 2*s

        # 方形平均
        t = a[::s2,::s2]
        t2 = (t[:-1,:-1] + t[1:,1:] + t[1:,:-1] + t[:-1,1:])/4
        tmp = a[s::s2,s::s2]
        tmp[...] = t2 + normal(0, scale, tmp.shape)

        buf = a[::s2, ::s2]

        # 菱形平均分两步，分别计算水平和垂直方向上的点
```

```
        t = a[::s2,s::s2]
        t[...] = buf[:,:-1] + buf[:,1:]
        t[:-1] += tmp
        t[1:]  += tmp
        t[[0,-1],:] /= 3 # 边上是 3 个值的平均
        t[1:-1,:] /= 4 # 中间是 4 个值的平均
        t[...] += np.random.normal(0, scale, t.shape)

        t = a[s::s2,::s2]
        t[...] = buf[:-1,:] + buf[1:,:]
        t[:,:-1] += tmp
        t[:,1:] += tmp
        t[:,[0,-1]] /= 3
        t[:,1:-1] /= 4
        t[...] += np.random.normal(0, scale, t.shape)

        scale *= d
    return a
```

观察图 19-14(右)中标记为 2d 的方格，我们发现菱形平均对应的方格无法用一个数组表示，因此程序中将它们分为水平和垂直方向上的两个数组分别计算。程序中大量使用数组切片，使得许多计算在 C 语言级别进行，从而提高程序的运算速度。如果读者觉得较难理解，可以修改一下程序：

- 注释掉随机数部分，并在迭代之前将数组 4 个角上的元素赋值为不为零的数值。
- 使用较小的数组，并且输出每次赋值之后数组 a 的值。通过观察数组 a 的变化，可以帮助理解程序和菱形方形算法。